Experimental Analysis of Algorithms for the Dynamic Graph Coloring Problem

Theunis, Menno

Award date:
2022

Department of Mathematics and Computer Science
Algorithms, Geometry & Applications

Experimental Analysis of Algorithms for the Dynamic Graph Coloring Problem

Master's thesis

Menno Theunis

25-07-2022

Supervision:
Marcel Roeloffzen

Assessment committee:
Marcel Roeloffzen
Wouter Meulemans
Yanja Dajsuren

Credits: 30

This is a public Master's thesis.

This Master's thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct.

Abstract

This thesis focuses on the dynamic graph coloring problem, a dynamic variant based on the well-researched graph coloring problem. This variant of the problem not only considers the number of colors used in the coloring for a graph, but also how many nodes in this graph need to change their color when the graph is changed. The balance between these two measures of quality, as well as running time, creates an inherent trade-off, in which algorithms solving this problem often only focus on one or the other. A variety of such algorithms already exist and are compared, as well as improved upon, in this thesis. Each of these algorithms uses different variables to measure its effectiveness, making it difficult to compare their advantages and disadvantages. Finding the right option for a practical application is thus unnecessarily difficult. By implementing the different algorithms and comparing them experimentally, we get a better insight of the strong and weak points of these algorithms. Using this knowledge we propose two new improved variants of these algorithms, obtained by combining aspects of the existing ones. We find that this approach of combining existing algorithms with different strong points often yields superior results and allows for a more versatile trade-off within the algorithm, making it suitable for a broader range of practical applications.

Contents

Chapter 1

Introduction

1.1 Background and motivation

The graph coloring problem is one of the most well known problems in computer science and combinatorics. The aim of this problem is to find a way of coloring some graph G such that no two adjacent elements have the same color. There are many variations of this problem, some of which consider different elements to color. The vertex coloring problem considers the coloring of the graph's vertices, only disallowing two vertices to have the same color if they are connected by an edge. The edge coloring problem, on the other hand, considers edges with colors and is restricted by whether two edges are connected to the same vertex or not. Both of these graph coloring problems can be extended to a weighted variant, where each element is assigned more than one color, none of which may be the same as any of the colors occupied by another connected element. The two problems can also be combined into the total graph coloring problem, in which both edges and vertices are assigned a color.

Regardless of the variation, one aspect generally remains the same: the number of colors used to color the graph in a valid manner is required to be as low as possible. To this extend, various goals exist when dealing with the graph coloring problem. Some approaches focus on finding the smallest number of colors the graph can be colored with, called the chromatic number, whereas other approaches attempt to find out whether or not a graph can be colored with a predefined number of colors.

Both of these problems have, however, been shown to be NP-hard, meaning there is no polynomial time algorithm available that solves it exactly. It is therefore that many research papers have been written about heuristic algorithms for the graph coloring problem, allowing some practical, if not exact, approaches for finding valid colorings with relatively few colors. These heuristics can be split into two categories: those that pre-order the vertices before coloring them according to this order and those that do not pre-order the vertices but rather decide which vertex to color next based on the partial coloring already created.

The graph coloring problem is interesting because it can be used to model several real-world situations, usually by following the general structure in which vertices represent items of interest, edges represent a binary relation between these items and

colors represent available resources. A few examples of real-world situations that can be modeled this way are timetable scheduling and frequency assignment. In timetable scheduling, for lectures at a university for example, vertices represent lectures, colors represent timeslots and edges are present between two vertices only if the lectures they represent are given by the same lecturers. When this graph is colored according to the rules of the graph coloring problem, lectures given by the same lecturer are thus assigned a different color and will not be scheduled into the same timeslot. Finding a coloring with the least amount of colors allows the schedule to use as few timeslots as possible. The frequency assignment case works in a similar manner, where vertices represent customers using a network, colors represent available frequencies and edges are present only if two customers are within a certain distance of each other, in order to avoid interference. The graph coloring problem, in it's various forms, can thus be used to solve various real-world problems.

While the majority of the research performed on this problem focuses on this static situation of coloring an entire graph from scratch. Other varieties have more recently come into focus that solve a related problem, but allow for different real-world situations to be modeled. Two such alternatives are the online, or streaming, graph coloring problem and the dynamic graph coloring problem. The online variant poses the problem of coloring a graph incrementally. The graph is updated over time by repeatedly adding a vertex and its edges to the vertices already present. A coloring is thus only a snapshot of one particular moment. Once a vertex has been colored in the online variation of the graph coloring problem, the color is fixed and cannot be changed. The dynamic graph coloring problem, set in a somewhat similar environment, where updates can consist of additions or removals of vertices or edges, differs in the way that it does allow for vertices to change color after already being assigned one during a previous update step, which allows the total number of colors required to be lowered by recoloring already existing vertices to a more optimal coloring. The dynamic variation still aims to keep these so called recolors to a minimum however, since changing the color of a vertex could have a high cost in the corresponding real-world problem. This thus creates a trade-off between the number of colors used and the number of recolors required. Depending on the situation the algorithms are applied in, recoloring vertices may or may not be possible and its cost might differ, making the choice between the streaming or dynamic variation and a preference for number of colors or number of recolors highly situational.

In this thesis, we will take a closer look at the trade-off presented by the dynamic graph coloring problem. We will do so in the context of unweighted vertex coloring, such that each vertex is assigned a color and edges are not. We believe this variation of the graph coloring problem is especially interesting in current times, where networks and data are constantly evolving to reflect the present situation. The dynamic graph coloring problem allows us the luxury of changing a vertex' color in a later update, even after it had already been assigned one, a welcome property when dealing with large ever-changing graphs.

Even though multiple approaches have already been proposed to efficiently solve this dynamic graph coloring problem, it remains unclear which one of them is best used in

which situation. The information provided about their performance are mostly limited to theoretical amortized bounds on running time, recolors and total number of colors used, and oftentimes these bounds include variables specific to each approach, making it difficult to compare them to each other. In this work we summarize the most popular approaches to solve the dynamic graph coloring problem and compare their effectiveness and trade-offs by running multiple experiments. We also propose two competitive new algorithms created by combining the existing approaches.

1.2 Contributions and organization

The main contributions this work provides are a clear comparison of the existing algorithms solving the dynamic graph coloring problem and the two new combination algorithms static-simple and DC-Simple. The general comparison of the algorithms aims at stimulating further research into different algorithms and into when each variation should be used. Currently it is difficult to find the correct algorithm to use for a project, since the papers introducing them do not clearly state the advantages and disadvantages as compared to the other available algorithms. In comparing these algorithms and running the experiments, the ideas for two new algorithms presented themselves, formed by combining some of the investigated algorithms together. These algorithms turn out to be quite competitive and provide a trade-off for some of the considered algorithms that do not normally allow for a parameter to control the importance of the number of colors versus the number of recolors. The implementations of these new algorithms, as well as any other files used during this thesis, can be found in the accompanying GitHub repository [1].

The rest of the work is structured as follows. Chapter 2 presents the terminology and prerequisite knowledge required to understand this thesis. In Chapter 3 each of the compared algorithms is introduced in short, after which Chapter 4 expands on the exact approach taken and specific implementation used for each algorithm. Chapter 6 introduces the experiments ran as part of this research and also includes the most important results. Finally, Chapter 7 and 8 draw conclusions based on the results and discuss what could be done to further this research.

1.3 Related work

The graph coloring problem, as summarized in [2], is a well studied problem. Many research papers have been dedicated to proving various aspects of this problem and to finding solutions for it. It has already been shown that finding the chromatic number of a graph [3] and finding out whether a graph can be colored with $k > 3$ colors [4][5] are both NP-hard problems. Despite this result, many practical algorithms have already been found for the static graph coloring problem that allow the coloring of graphs using heuristics that oftentimes result in using fairly few colors [6][7]. Many of such heuristics depend on first ordering the vertices and then coloring them one by one in the selected order. It has been proven that there must exist at least one such ordering of vertices

for every graph that results in an optimal coloring, but finding this ordering is still an NP-hard problem. The first and most well known such ordering is based on the degree of the vertices and described in [8].

Apart from the static graph coloring algorithm, the online and dynamic graph coloring problems have been studied to a lesser extend as well, as such there are papers presenting algorithms for the online variant of the graph coloring problem [9], papers outlining a comparison between the different online algorithms [10], and papers presenting a similar comparison for the dynamic variation of the problem, but focusing on theoretical upper bounds and biparite graphs [11].

In this work we will expand this source of knowledge by presenting an experimental analysis for some of the most interesting and popular algorithms for the dynamic graph coloring problem and comparing their results. The algorithms we will consider are those proposed by Bhattacharya et al. [12], Barba et al. [13], Solomon et al. [14] and Yuan et al. [15], as described in Sections 4.2, 4.3, 4.4 and 4.5 respectively.

Chapter 2

Preliminaries

Central to this work is the idea of *colorings*, being an assignment of colors to the vertices of a graph. Such a coloring is *valid* if all vertices have been assigned a color and no edge has two endpoints with the same color. Such a valid coloring uses a limited set of distinct colors, which we will call the *total number of colors used*. This total is lower bounded by the *chromatic number* C, which is the optimal, or lowest, number of colors required to get a valid coloring for a graph. Note that finding the chromatic number is not always viable, as the graph coloring problem is NP-hard. In this work we therefore approximate C by running a static greedy coloring algorithm on a graph which does not guarantee, but is likely to produce a 'good' coloring with close to C colors. The particular static greedy coloring we use is the one described in [8] and detailed in Section 4.1 of this thesis, and has been chosen for its simplicity and popularity in research papers and graph libraries.

For ease of implementing the algorithms and talking about different colors we occasionally refer to colors as if they are numbers: green, red or blue may be represented by 1, 2 or 3. As long as the color values are different for two adjacent nodes the coloring is correct. With this idea of numbers representing colors comes the idea of a *lowest color*, being the color that has the lowest number associated with it, this definition will allow algorithms to find an unoccupied color more efficiently, and helps guarantee that the set of colors being used remains small.

Apart from colorings, this work will largely focus on the number of times a vertex that was already assigned a color is assigned a different color in order to resolve *conflicts* or improve a coloring. This process of changing the color of a vertex will be referred to as a *recolor*, and a conflict is defined as two adjacent nodes, connected by an edge, having the same color and thus violating one of the properties that makes a coloring valid. If a conflict occurs at least one of the involved vertices needs to be recolored by an algorithm to ensure a valid result.

With these definitions we can define the dynamic graph coloring problem considered in this work more precisely: The input of the problem will consist of two parts, the *initial graph* and the *update sequence*. The initial graph will consist of all nodes required at any point during the update sequence and a set of edges forming the initial connections. The initial graph is assumed to start with a good coloring, which we generate using the static

greedy algorithm from Section 4.1. The second part of the input, the update sequence, is represented by a list of tuples, each tuple consisting of an edge and a Boolean stating whether that update corresponds to the *addition* or *removal* of that edge. We ensure that updates are always possible, and thus do not allow for edge insertions of edges already present, or between non-existent nodes, we also do not allow for edge removals of edges not present in the graph at that point in time. Note that while vertex additions and removals could also be modeled in these update sequences, these actions are of little interest in the algorithms considered here and are thus omitted. This is because the addition of a vertex with edges can be simulated by first adding a disconnected vertex and adding the corresponding edges using normal edge additions after. The removal of a vertex with edges can be simulated by using regular edge removal actions before removing the disconnected vertex. Since the removal or addition of a disconnected vertex is trivial and the edge removal and addition actions are already defined, we do not explicitly define these vertex related actions. Also note that, while the update sequence may be generated in full before the start of an experiment, the algorithms only receive these updates one at a time and do not have any knowledge on which, or even how many, updates will follow.

With this input the goal of the problem is for the algorithms to generate and output a valid coloring for each of the states the graph is in after executing an update from the update sequence. The average number of colors used and average number of recolors required are saved, as well as the time spent on generating all the colorings. These are values the algorithms will be compared on, and the aim is thus to achieve colorings with as few colors as possible, while also avoiding recolors whenever possible. These two opposing factors are the cause of an inherent trade-off many of the algorithms are capable of making.

This trade-off that is present in many of the algorithms we consider is often guided by a *parameter* that can be adjusted to focus more on the number of colors used or on the number of recolors. These parameters differ from algorithm to algorithm, and some of the algorithms do not provide one at all.

Some additional important variables are N, representing the number of nodes in a graph and Δ, representing the *maximum degree* of a graph at some point in the update sequence. The maximum degree can change during an update sequence by adding or removing edges, giving this variable different values over time. It is because of this behaviour and the fact that keeping track of these changes can be quite inefficient, that we decide to not use Δ in the algorithms, but rather replace it with δ where necessary, with δ representing the *local degree* of a node in the graph. While Δ is often used in bounds for the number of colors or recolors in graph coloring algorithms, the algorithms considered here actually also function properly by using the local degree δ. We thus decide use δ in the algorithms, but keep Δ as an upper bound to reason with.

Since the maximum degree Δ is simply the maximum of all local degrees δ, this substitution does not create any problems and allows us to continue to bound certain algorithms using Δ as a variable.

Finally, some algorithms make use of *black-box algorithms*, meaning the algorithm

will use a different graph coloring algorithm to create a coloring for some (sub)graph, of which the result can later be used to create a more complete or efficient coloring. The input and output of such a black-box algorithm are accessible to the overarching algorithm but the fact that it is a black-box means the inner workings of the algorithm being used as a subroutine cannot be influenced, changed or observed. This also means that any algorithm with the correct input and output can be used as such a black-box, if a more efficient algorithm is found, the algorithms currently used as black-boxes can be swapped out for the more efficient variant, making the overarching algorithm more efficient as well without having to redesign anything.

Chapter 3

Introduction to the Algorithms

In this chapter we introduce the algorithms by assigning them a unique name, providing an intuitive notion of their workings, stating known asymptotic bounds and presenting their potential strengths or weaknesses.

Static Greedy Algorithm While the static greedy algorithm, as described in [8] and [7], will not have an explicit presence in the experiments, it is worth mentioning as it is the oldest and simplest of the static greedy algorithms that is commonly used. In this thesis it will serve as a baseline for all dynamic algorithms. An evolving graph could be recolored by simply recalculating the entire coloring using a static algorithm such as this one. This approach would be very inefficient when it comes to number of recolorings and running time, but would provide a high quality coloring, with few colors, almost every time. We thus use this algorithm as a suitable baseline for the total number of colors used when comparing the dynamic algorithms considered in this thesis. These dynamic algorithms are likely to be much more efficient when it comes to recolors and running time but are unlikely to achieve the same total number of colors. The static greedy algorithm works by assigning a priority to each vertex based on their degrees, and assigning the vertices a color in a specific order, as further described in Section 4.1. This algorithm is also used whenever another algorithm requires a static black-box algorithm as a subroutine.

Random Warm-Up This randomized coloring algorithm, as further described in Section 4.2 is a combination based on the two warm-up results from [12]. While it is not the main algorithm presented in the paper, it does achieve the same bound on colors used, namely $\Delta + 1$ colors where Δ is the maximum degree in the graph at the time of the coloring. The random warm-up algorithm used in this thesis randomly assigns a 'free' color to a vertex that needs to be recolored, where a free color is defined as a color not occupied by one of the vertex' neighbors. Such a free color always exists, making this the simplest and fastest algorithm discussed here. It additionally uses only very few recolors, due to recoloring at most one node per update and low probability of causing conflicts. This random warm-up algorithm is also the algorithm that is used whenever another algorithm requires a dynamic black-box algorithm as a subroutine.

Small- and Big-Bucket Algorithms The small- and big-bucket algorithms as described in [13] and Section 4.3, divide the vertices of a graph into different sets of buckets, each with their own color palette. Each bucket uses a static black-box algorithm to color its subgraph and the final coloring is obtained by combining the colorings of all buckets. These algorithms use a parameter d to manage the trade-off between number of recolors and number of colors used. This parameter should lie within the range $[1..\log N]$, with N the number of nodes in the graph. A high value of d causes the trade-off to be more balanced, whereas a low value of d makes the algorithms skew further to one of the extremes. The small-bucket algorithm favors fewer recolors and manages to only use $\mathcal{O}(d)$ amortized recolors per update, while using $\mathcal{O}(dN^{1/d}C)$ colors, with C being the chromatic number. The big-bucket algorithm favors fewer total colors used and achieves $\mathcal{O}(dC)$ total colors while having $\mathcal{O}(dN^{1/d})$ amortized recolors per update. The small- and big-bucket algorithms produce more similar results as d increases up until they converge at $d = \log N$. The small- and big-bucket algorithm are therefore one of the more versatile options when it comes to providing a trade-off, as the combination of these two algorithms allows coverage of the whole spectrum in the trade-off between recolors and number of colors used.

Static-Dynamic Algorithm The static-dynamic algorithm is described as the algorithm for general graphs in [14] and further detailed in Section 4.4. This algorithm uses a parameter l to manage its trade-off, and revolves around using some dynamic graph coloring algorithm to resolve conflicts for l update steps before running a static coloring algorithm on an intelligently selected subset of nodes in order to improve the quality of the coloring. The amount of steps l before a static black-box step is given as a parameter. The authors claim to achieve bounds of $\hat{O}(\frac{C}{\beta}\log^2 N)$ total colors and $\mathcal{O}(\beta)$ expected recolors per update, where the total color bound suppresses polyloglog(N) factors and $\beta = \frac{\log N}{l}$. The variable N representing the number of nodes in the graph. To obtain the coloring produced by the algorithm each vertex is assigned a tuple consisting of its dynamic and its static color (c_1, c_2). Since these tuples are then viewed as colors themselves, the total number of colors used in this algorithm is comparatively high.

DC-Orient The DC-Orient algorithm as described in [15] and detailed in Section 4.5 does not focus on the number of recolors per update, but rather aims at simulating the greedy static algorithm from Section 4.1 in a dynamic manner. The algorithm works by generating a priority ordering among the vertices based on their degrees and creating a directed version of the graph to use internally. When a conflict arises the vertex with higher priority is allowed to keep its color and the lower priority vertex must change its color to one not occupied by one of its in-neighbours in the directed graph. When a vertex changes color it also recursively recolors its out-neighbors with lower priority if necessary. By updating the colors in this way the coloring and thus also the total number of colors used is generally identical to the one that would be generated by the greedy static algorithm. As a trade-off, however, the number of recolors per update and running time are both relatively high when compared to the other dynamic algorithms.

Chapter 4

Implementation Details

In this chapter the workings of the algorithms are expanded upon to allow for easier understanding and reproduction of the implementations used in this thesis. Each algorithm is described textually as well as using pseudocode if none was present in the cited paper.

4.1 Static Greedy Algorithm

The static greedy algorithm used in this thesis is the greedy coloring algorithm provided by the NetworX Python library [16], which corresponds to the Greedy-Color method described in [7].

It assigns each node a priority based on its degree, in which higher degree nodes gain higher priority, and arbitrarily orders the nodes with identical degree. Using this priority ordering the algorithm colors the nodes one by one, starting with the node that has the highest priority. Whenever a node is colored in this way, it is assigned the lowest color that does not cause conflicts on any of its edges. Since any node in the graph can have at most edges equal to the maximum degree Δ, there can be at most Δ colors that are already occupied by a node's neighbour. Each node can therefore be colored using a color value of $\Delta + 1$ or lower.

4.2 Random Warm-Up

The random warm-up algorithm stems from [12], in which three variations of a randomized coloring algorithm are proposed. The first and second of which are warm-up results leading up to the main algorithm presented by the authors. Each of these variations has its advantages and disadvantages, and for this thesis a combination of the first two will be used.

In the first warm-up result, the authors propose a simple algorithm that uses randomness to quickly and simply recolor any node involved in a conflict. When such a conflict occurs, this warm-up result picks one of the two nodes involved in the conflict, more specifically the one that has been recolored most recently, and generates two sets of colors: one set containing 2Δ colors, the complete color palette for the graph, and

another set containing only the colors occupied by the neighbours of the node to be recolored. The algorithm then computes the difference between these two sets and obtains a set of colors that are not yet occupied by the node's neighbours and could thus be assigned to the node without creating a new conflict. The algorithm decides on one of these colors uniformly at random and assigns the node this new color, resolving the conflict. Unlike in the static greedy algorithm from Section 4.1, a color palette of 2Δ rather than $\Delta + 1$ is decided on here, in order to prevent a malicious adversary from providing updates that force the algorithm into choosing only a single free color. By making the color palette larger, the algorithm will never be predictable and an adversary is thus unable to force the coloring of the algorithm in any direction.

The second warm-up result provided in the paper suggests a different way of avoiding predictability. This variation of the algorithm does use a color palette of $\Delta + 1$, but also allows a node to be assigned a color already occupied by its neighbor, provided that only a single one of its neighbors currently uses that color. By selecting the new color in this way, the set of colors to randomly choose from is always larger than one, and the behaviour of the algorithm therefore never predictable. It does, however, potentially create new conflicts during the assignment of this color, and resolves these by recursively recoloring the neighbor with the chosen color in the same manner. This way of resolving conflicts could lead to a long chain of recursive recolors and is difficult to analyse because of it. The number of colors used in this variation is lower than in the first random warm-up, however.

The final variation of this algorithm presented in [12] is the authors' main contribution. The approach is very similar to that of the second warm-up, differing only in the way the nodes are structured. By using a leveled structure of nodes, the main algorithm becomes more difficult to implement, but easier to analyse. The authors state that by only recursively recoloring nodes with a lower level, it is possible to bound the number of recursions and prove that termination is eventually reached while still retaining the small color palette of $\Delta + 1$ and unpredictability of the algorithm.

Because we will not consider a malicious adversary with knowledge and the ability to affect the update sequence, however, in this thesis we opt for a combination of the first and second warm-up results. The algorithm called the random warm-up in this work, will function almost identically to the first warm-up from [12], while only using a color palette of $\Delta + 1$. This allows for a simple and efficient implementation, while retaining the lower bound for number of colors used. Even though the unpredictability of the algorithm is therefore partially sacrificed, this fact is irrelevant for the experiments performed in this thesis and outweighed by the advantages of having a simple and efficient implementation.

Additionally, because the remaining algorithms considered in this work do not have knowledge about the maximum degree Δ, we will instead use the adjustment proposed in [12] for using the local degree rather than the maximum degree. The difference caused this way is that the palette of available colors when a node is recolored is as large as $\delta + 1$ rather than $\Delta + 1$, with δ being the degree of the node to be recolored at that point in the update sequence. This change does not affect the upper bound of $\Delta + 1$

colors used in total.

It is worth noting that during the implementation phase of this thesis some preliminary experiments were performed on different versions of the algorithms in order to come to a suitable final variant. A small comparison between this final random warm-up and the second warm-up result from [12], based on both maximum degree Δ and local degree δ is provided in appendix A, Section A.1.

4.3 Small- and Big-Bucket Algorithms

The small- and big bucket algorithms as proposed by Barba et al. in [13] are two complementary algorithms that allow for a trade-off between recolors and total number of colors used. Central to both of these algorithm is the idea of vertices being put in different buckets and each bucket executing a static black-box coloring algorithm on the subgraph consisting of the vertices in that bucket. Since each bucket uses its own independent color palette, if each bucket is validly colored this means the union of all these colorings is also a valid coloring for the entire graph.

The complementary aspect of the algorithms is that, depending on a parameter d, these algorithms can achieve different ranges of the trade-off between recolors and total number of colors used. The small-bucket algorithm favors fewer recolors and uses only $\mathcal{O}(d)$ amortized recolors per update, but uses $\mathcal{O}(dN^{1/d}C)$ colors, with N being the number of vertices in the graph and C being the chromatic number. The big-bucket algorithm skews the other way and favors fewer total colors, using only $\mathcal{O}(dC)$ colors but requiring $\mathcal{O}(dN^{1/d})$ amortized recolors per update. As parameter d increases, the small- and big-bucket algorithms produce more similar results up until they converge at $d = \log N$, after this point, higher values for d will have no effect.

The overall structure of the small- and big-bucket algorithms consists of a set of buckets divided over different levels. When a node is added to one of these buckets, that bucket's coloring is recomputed using the static black-box coloring algorithm. If the buckets on a level are deemed to be full, they are all grouped together and moved into the first bucket on the next level. The sizes of the buckets as defined in the algorithm ensure such a transfer is always possible. After moving up a level, the new bucket will statically recolor its nodes. This process of adding vertices to buckets and moving nodes from lower to higher levels continues until the end of the leveled structure is reached and a full reset occurs.

Depending on the parameter d the algorithms will use more or fewer levels of buckets, each with different capacity: in the small-bucket algorithm there are d levels (0..d-1) of relatively small buckets where each level has s buckets with capacity s^i, where i is the bucket's level, $s = N_R^{1/d}$ and N_R is the number of nodes in G at the moment the buckets are created. The big-bucket version of the algorithm also has d levels (1..d) of buckets, but each level has only a single bucket of capacity s^i. Additionally, both versions of the algorithm have a final level with only one bucket that has no limit on its capacity. This final bucket is the overflow or reset bucket. A visual representation of the bucket levels and each bucket's capacity is shown in Figures 4.1 and 4.2

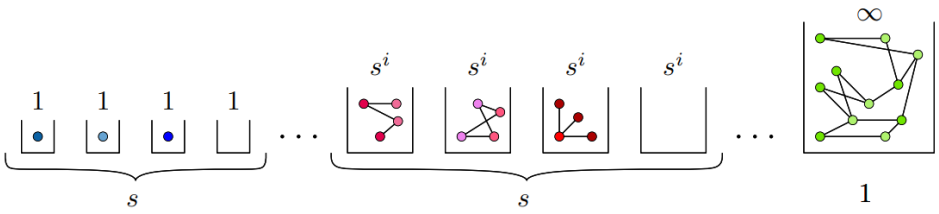


Figure 4.1: Visual representation of the buckets used in the small-bucket algorithm. Figure taken from [13].

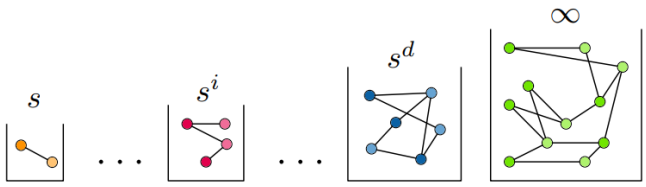


Figure 4.2: Visual representation of the buckets used in the big-bucket algorithm. Figure taken from [13].

Because of the way the bucket levels and capacity are defined, the nodes from all buckets at level $i-1$ can fit into only a single bucket at level i. Both variants of the algorithm have some invariant to guarantee there is enough space to move vertices from one level to the next: the small-bucket algorithm has a *space invariant* stating that each level always has at least one empty bucket and the big-bucket algorithm has a *high point invariant* stating that each level contains at most $s^i - s^{i-1}$ vertices. These invariants allow the algorithm to always have enough room to move all vertices from a certain level to a single bucket on the next level.

Finally, the algorithms handle updates as follows. Initially all vertices are in the overflow bucket and a static coloring is generated using the black-box algorithm on the entire graph. Whenever an edge is added to the graph, one arbitrary endpoint of this edge is removed and reinserted into the graph in order to simulate a vertex addition update. Whenever a vertex is added to the graph it is put into a bucket on the first level: in the small-bucket algorithm the *space invariant* guarantees there is at least one empty bucket in the first level and in the big-bucket algorithm the *high point invariant* guarantees the bucket on the first level has enough room left for an additional vertex. If these additions to the buckets invalidate the corresponding invariant, all vertices on the lowest level are moved to the first empty bucket on the next level or added to the only bucket on the next level, for the small- and big-bucket variants accordingly. At the second level this movement of vertices may have invalidated the invariant and the process is repeated until all levels satisfy the invariant or until the reset bucket is reached. In the first case, each bucket that changed contents recomputes its coloring using the static

black-box algorithm for the subgraph consisting of the vertices in that bucket. In the second case, the entire graph is recolored using the static black-box algorithm and all buckets are removed and created again with an updated value for $s = N_R^{1/d}$.

While edge and vertex removals are not explicitly mentioned by the authors, we extend these algorithms to allow for these in the simplest way possible. When an edge is removed, the graph representation is simply updated without changing the contents of the buckets and when a vertex is removed, we simply remove it from whichever bucket it was in without running the static black-box algorithm again for that bucket. This way of handling removals will never invalidate either of the invariants, nor will it create conflicts within the coloring of the graph.

The pseudocode used to implement both the small- and big-bucket algorithms can be found in algorithm 1 and 2, located in appendix B.

4.4 Static-Dynamic Algorithm

The static-dynamic algorithm for general graphs as described by Solomon et al. in [14] aims at combining static and dynamic black-box algorithms in an intelligent way that allows for a trade-off between the advantages of both. In order to achieve this the algorithm uses two representations of the same graph: the full graph G as also used by other algorithms for use with the static black-box algorithm and a sparse variant G' of this graph with the same vertex set but only very few edges for use with the dynamic black-box algorithm. Both of these graphs will have a coloring for all vertices as provided by the black-box algorithm used on that graph. The final coloring the static-dynamic algorithm produces for each vertex is a combination of both of those colors, in the form of a tuple (c1, c2).

After each update at least one of the black-box algorithms is run on a relevant part of the graph in order to ensure conflicts are resolved. It is worth noting that if a conflict is solved in one of the black-box algorithms it is not necessary to also solve it in the other, since the combined tuples (c1, c2) of the conflicting vertices will already be different and there is thus no conflict in the final coloring. The decision of when to run the static and when to run the dynamic black-box algorithm and on what subgraph to run them exactly depends on two central concepts: update segments and the recent degree of vertices.

In this static-dynamic algorithm the entire sequence of updates is split into update segments of length Nl with N being the number of nodes in the graph and l a parameter used to guide the trade-off in the algorithm. One update segment of size Nl is defined to be at level 0 and a leveled structure is created by splitting the segments of each level into two for each next level. This causes level $\log N$ segments to have length l and in general causes level i segments to have length $Nl/2^i$. The authors state that for ease of use, update segments with identical endpoints are removed until only the one at the lowest level remains. A visual representation of the created segments can be found in Figure 4.3. If the update being executed at some point in time is part of an update segment at some level in this structure, we call this level *active*, if no update segment covers the current update on a level, this level is considered *inactive*.

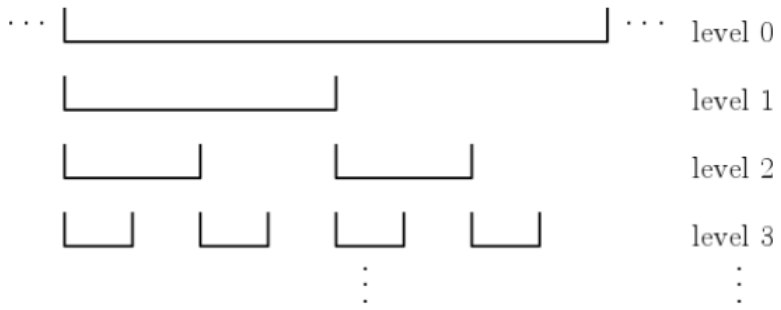


Figure 4.3: Visual representation of the update segments used in the static-dynamic algorithm. Figure taken from [14].

Depending on the parameter l the trade-off this algorithm makes can be influenced. The authors claim to achieve bounds of $\hat{O}(\frac{C}{\beta}\log^2 N)$ total colors and $\mathcal{O}(\beta)$ expected recolors per update, where the total color bound suppresses polyloglog(N) factors and $\beta = \frac{\log N}{l}$. Higher values of l should thus intuitively lead to fewer recolors but potentially more total colors used.

Apart from separating the updates to be executed into different segments, this leveled structure also keeps track of which nodes will be involved in a static recoloring step later. To this end, each level will be assigned a node set, initially empty, that represents the nodes to be recolored statically whenever the update segment on that level ends. Which nodes are assigned to these node sets depends on their *recent degree*, an integer representing how many edges have been added to a node since it was involved in a static recoloring.

When an update is executed, both graphs G and G' are updated to reflect the change. Vertices involved in an edge addition also keep track of their new recent degree. The recent degrees of all vertices start at 0, and get increased whenever an edge is added to a node. If after an update a node that is not yet in an active level's node set has the highest recent degree, it is added to all active levels' node sets it is not yet part of, ensuring this node will be involved in the static recoloring that occurs at the end of those levels' update segments.

It is worth noting that the nodes part of these node sets are not well defined in [14], only stating that nodes with highest recent degree at the end of an update interval or subinterval should be part of it, without clarifying how such a subinterval is defined. The way we handle these node sets assumes a subinterval can have any size smaller or equal to an update segment, but always starts at the same point the update segment it is a subset of does. This way of defining a subinterval falls within the general scope of the term used in [14].

If after an update the end of any update segment in the leveled structure has not

been reached, potential conflicts are handled by the dynamic black-box algorithm by running it on (the most recently added edge in) G'. If, however, the endpoint of one of the update segments has been reached, a static black-box algorithm is ran on the subgraph consisting of the vertices in the node set corresponding to the level of that update segment, after which this node set is emptied. This static recoloring of such a subgraph is defined to use a color palette unique to the level of which the update segment has just ended. When a static black-box algorithm is ran, all chosen vertices reset their recent degree to zero and all edges adjacent to any of these vertices are removed from G'. If a conflict still occurs after the static recoloring of the chosen subgraph, for example when an edge is added between two vertices with the same color, but neither of them have the highest recent degree and are thus not included in the static recoloring, then a dynamic recolor still occurs based on the updated G'. By running the updates in this manner conflicts are always resolved by either the static or the dynamic black-box algorithm.

Note that, while G and G' are immediately updated when an update occurs, their coloring remains the same until a static or dynamic black-box algorithm is triggered. Intuitively this seems reasonable but in practice we find that updating G' will automatically trigger the dynamic black-box algorithm to also update its coloring. Because this is not desired behaviour the black-box dynamic algorithm, in our case the random warm-up from Section 4.2, is adjusted slightly to also allow for adding edges without triggering an update to the coloring. By doing this the dynamic algorithm is officially no longer a black-box. Since this is how the original paper [14] defines these terms, however, and since it improves readability, we opt to keep this terminology in place regardless.

When the endpoint of the update segment at level zero is reached, all vertices are recolored using the static black-box algorithm and new update segments are generated. From this it follows that all vertices also reset their recent degree to zero and that G' clears all its edges. Note that the authors do not mention resetting the dynamic colors present in G' at this point, which results in using more combined colors (c1, c2) than necessary after such a full reset of the static coloring of G occurs. An alternative version of this algorithm that does reset the dynamic graph as well has been implemented too. A small comparison between this variation and the main version of the algorithm can be found in appendix A, Section A.2.

The pseudocode for the static-dynamic algorithm can be found in algorithm 3, located in appendix B. Note that we do not presume to know all update steps in advance and thus do not explicitly generate the 'update segments' in advance, but rather keep track of our position within the leveled structure by using counters.

4.5 DC-Orient

The final algorithm considered in this work is DC-Orient as described in [15]. What sets this algorithm apart from the rest is its aim to simulate the static greedy algorithm discussed in Section 4.1 in a dynamic manner. The authors observed that when a colored graph is recolored using the static algorithm after one update occurred, only few of the

vertices change color in many of the cases. This observation is the only attempt at reducing the number of recolors in this algorithm, as there is no parameter available for a trade-off. It thus heavily leans towards a small total number of colors used and is practically the opposite of the random warm-up algorithm discussed in Section 4.2.

DC-Orient does not provide a guarantee on the number of recolors but does achieve the same coloring as the static greedy algorithm, which could be bounded with an upper bound of $\mathcal{O}(\Delta + 1)$, with Δ the maximum degree in the graph, although this bound is rarely accurate, considering DC-Orient aims at achieving a total coloring with close to C colors. The difficulty in bounding the number of colors used stems from the fact that the chromatic color C is also difficult to bound, and heuristic algorithms such as the greedy static coloring algorithm DC-Orient was based upon are actually used to upper bound this chromatic number. An accurate bound for this approach is thus unavailable.

The algorithm works by defining a priority ordering based on the degree of the vertices in the graph, much like the static greedy algorithm, where high degree nodes get priority over low degree nodes. A directed graph G* is generated in which the edges of G are pointed from high priority to low priority nodes as can be seen in Figure 4.4.

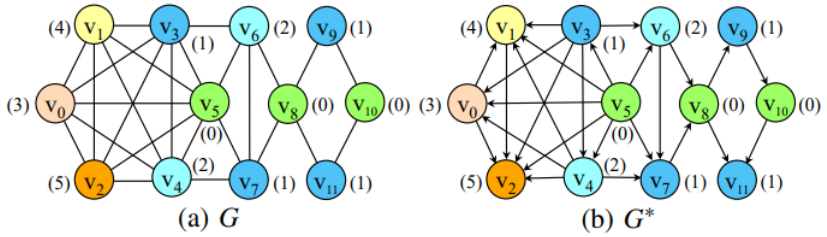


Figure 4.4: Example of a directed graph G* created and used by DC-Orient. Figure taken from [15].

When an update occurs G* is first updated to reflect the new priority ordering of the vertices and potential conflicts are then resolved by executing a CAN step in which the vertex of highest priority involved in a conflict first *collects* the colors not occupied by it's in-neighbors in G*, then *assigns* itself the 'lowest' color from the set of available colors and finally it *notifies* it's out-neighbors in G* that they might have to do a CAN step as well, in case a color was chosen that one of these lower priority nodes had assigned to them. Because G* is directed and acyclic this chain of CAN steps is guaranteed to finish.

Because of the way the authors structured their paper [15], many different versions of DC-Orient exist. The basic version of the algorithm is the easiest to understand, but is very slow in practice. The authors therefore introduce different ways to make their algorithm more efficient. One of these additions is a Dynamic In-Neighbor Color Index (DINC-Index) to keep track of the colors of each node's in-neighbors. By maintaining this DINC-Index the algorithm no longer needs to access each node's neighbors to assign

it a new color, which in practice turns out to be much more efficient. Furthermore, some pruning strategies are suggested to avoid unnecessary recursions to nodes that do not cause conflicts. Both the basic and the optimized versions of the algorithm have been implemented, but since the results, apart from the running time, are identical we have decided to use the optimized version in our experiments. A small experiment to compare the running times of both versions can be found in appendix A, Section A.3.

The pseudocode for the basic algorithm, as well as for all the extensions, can be found in the original paper [15].

Chapter 5

Additional Algorithms

In addition to the four classes of algorithms taken from the relevant papers, we have implemented two combinations of these algorithms that can overcome some of the issues the original versions face.

5.1 Static-Simple

When testing the algorithm implementations, it becomes apparent that not all of them perform as well as one would expect. For example, one would expect the results from the static-dynamic algorithm to fall somewhere in between the results of the random warm-up and DC-Orient, considering these are the extremes when it comes to a focus on low average number of recolors or low total number of colors used. During preliminary experiments it became apparent that, unlike other algorithms, the static-dynamic approach does not seem to fall into this expected range, but instead performs objectively worse in many cases. More specifically, the number of colors used was significantly higher than the alternative algorithms for almost all values of parameters. We hypothesise that the main reason for this issue is the multiplication in total number of colors used that occurs when combining the static coloring and the dynamic coloring into a single combined color (c1, c2). We therefore introduce a simpler version of the algorithm named static-simple. The static-simple algorithm is identical to the static-dynamic algorithm when it comes to the static black-box component. Static black-box algorithms are still ran at the same moments and on the same subgraphs. The difference with static-dynamic is that there is no sparse graph G' on which a black-box dynamic algorithm is ran, instead, we simply use the same approach as the random warm-up algorithm from Section 4.2, when a conflict occurs that is not solved by any static black-box executions, simply pick one of the conflicting vertices and assign it a random color from the set $\{0, .., \delta\}$ that is not occupied by any of its neighbors yet. Such a color always exists, and by handling dynamic conflicts in this manner we prevent the need of creating combined colors (c1, c2). The new bound on colors used thus becomes $\mathcal{O}(\Delta+1)$, as both black-box algorithms use at most $\Delta + 1$ colors. Additionally, this randomized process for solving conflicts is very similar to that used in static-dynamic, considering the random warm-up from Section 4.2 is used as the dynamic black-box in that algorithm.

5.2 DC-Simple

The second new combination of algorithms aims at creating a trade-off between the two extreme algorithms DC-Orient and the random warm-up. These algorithms both provide excellent results in one aspect, but less than desirable results in the others because both focus strongly on either recolors or number of colors used, without much regard for the other aspect and no trade-off to allow for any nuance. The random warm-up manages to handle updates with very few recolors in a very short time, but uses a lot of colors in the process. DC-Orient on the other hand achieves a high quality coloring with a low number of colors but needs many recolors and takes long to run. By combining the two algorithms and adding a parameter to control which of the two to use, we allow a trade-off between recolors, number of colors and running time. The resulting algorithm is DC-Simple, which combines DC-Orient and the random warm-up algorithm in the simplest way possible. A parameter p between 0 and 1 is added to the algorithm, representing the probability of taking a random warm-up step rather than a DC-Orient step. If an update occurs the algorithm decides randomly with weight p whether to do a random step or a DC-Orient step. A random step uses the same approach as in static-simple: one of the conflicting nodes is assigned a random color from the set $\{0, .., \delta\}$ not occupied by one of its neighbors. When a DC-Orient step is executed, the algorithm does exactly what DC-Orient would do, and thus intuitively 'overwrites' some of the randomly chosen colors in its CAN step chain. We thus expect the total number of colors used to remain quite low, while the average number of recolors should be reduced. Note that in the optimized version of DC-Orient the DINC-Index needs to be maintained whenever an update occurs, in the optimized version of DC-Simple we therefore always maintain this DINC-Index, even if a random step is taken. A small comparison between the basic and optimized versions of DC-Simple can be found in appendix A, Section A.4.

We note that both the random warm-up algorithm and DC-Orient have the same bound on total number of colors used, namely $\mathcal{O}(\Delta+1)$. By combining these algorithms, knowing neither algorithm will ever assign a vertex a color outside of the color palette $\{0, .., \Delta\}$ we can conclude that DC-Simple will also use at most $\mathcal{O}(\Delta+1)$ colors, although likely much fewer, since this bound is not tight for DC-Orient. The expected number of recolors per update is more difficult to bound because neither of the original algorithms state a clear upper bound on the number of recolors. We can tell from the way the random warm-up works that at most a single node is recolored in each update step and could thus bound the recolors per update for a random step by $\mathcal{O}(1)$. The recursive nature of the DC-Orient CAN steps make it difficult to bound the expected number of recolors per update for that algorithm, which results in difficulties bounding DC-Simple recolors as well.

Chapter 6

Experiments

This chapter provides a look at the experiments performed during this research. Its aim is to measure the performance of each algorithm in different situations and to test some hypotheses formed in the implementation phase of the algorithms. Each experiment will consist of an explanation, motivation, results and discussion.

Most experiments consist of a generated graph and update sequence. The graph will function as a starting point and the update sequence will provide updates to this graph. The implementation of these allows us to generate graphs and update sequences with various properties. The graph can have a predetermined number of nodes or edges and allows for different distributions of the degrees in the graph. More precisely, the most basic graphs generated are simple random graphs with a set number of vertices and a parameter for edge density. This density parameter represents the probability that an edge from the set of all potential edges between all nodes is actually present in the graph. Additionally, the generation of these graphs allows for a skewed distribution of edges, in which nodes are assigned a priority and edges adjacent to nodes with a higher priority get a higher probability of being added to the graph. The strength of this effect can be adjusted and thus allows for experimenting with the effect of this bias on the different algorithms. A more detailed description of the implementation of this skewed edge selection can be found in Section 6.5.

Two separate methods of generating update sequences are used: increasing and stream based. For the increasing update sequence we generate a sequence of updates that consists of edge additions only, we start with an empty graph that only has vertices, obtained by removing edges from the initially generated graph, and add one edge back at a time until the final generated graph has been built. The update sequence, in this situation, thus consists of the edges that were originally present in the generated graph. This variation allows for the edge additions to occur in a randomized order or in a more organized 'expanding' or 'node focused' order. The expanding order allows us to simulate a breadth-first-search type behaviour, in which the edge insertion updates are ordered in such a way that edges adjacent to the connected component so far are added first with higher probability. Similarly the node focused approach assigns each node a priority and adds edges adjacent to nodes with higher priority first, this means the updates will likely be concentrated on one node at a time. More details about the

expanding and node focused approaches can be found in Section 6.6.

Alternatively we can generate an update stream consisting of both edge insertions and deletions, in this case we start with the generated graph and add and remove edges with the same probability, such that the number of edges remains largely the same throughout the update stream. The edges that are removed over time can be picked at random or have an increased probability to be removed as they remain in the graph for a longer period. These two options are given the names random stream and decaying stream. Note that in the stream variant of the experiments, the generated graph, including its edges, functions only as a starting point. The edge additions or removals in the update sequence are generated after the starting graph has been generated, and uses either the random or decaying approach. More information on the decaying approach is presented in Section 6.4

Optionally the generation of both the graph and update sequence allow for some randomized 'variation' such that two graphs or sequences generated with the same parameters will result in similar but different instances. It is therefore possible that graphs intended to fall into the same range, size or density-wise, slightly differ from each other. This slight variation should help reduce the chance of a recurring outlier. Each experiment will clearly state the size and density of the graph and with which biases the graph and update sequence were generated. Every experiment will also provide an estimate for the average chromatic number C, obtained by running the greedy static algorithm on the graph after each update and counting how many colors are used on average.

The generated experiments described in this work consist of a set of experiments covering the effect of variations to the input graph on the algorithms, such as number of nodes, density or degree distribution within the graph. Furthermore, various experiments provide an insight on the effect that different orders of an update sequence can have on the algorithms, checking both the performance on random order and a more node-focused order of the same updates. Finally, a stream based update sequence is experimented with, in order to find out how the algorithms function during extended use in a fairly stable environment. This stream experiment is ran on both random and decaying update sequences, in which edges are likely to be removed as they become older.

Apart from these generated graphs and update sequences we also consider a real world dataset. This dataset, as found in [17], models the hyperlinks between various subgroups called 'subreddits' on the social media platform Reddit. This dataset, contrary to most generated experiments, provides us with some real-world properties and biases, such as being much larger, but also rather sparse around some nodes and highly concentrated around others. The aim of this final experiment is to see whether the results obtained from the generated graphs are representative for real-world usage of the algorithms, and otherwise, what differences occur in their performance.

All algorithms were implemented in Python 3 using the NetworX library [16]. The experiments were executed using the notebook functionality in Visual Studio Code on a system with an octacore Intel i7-6700K CPU at 4 GHz and 16 GBs of RAM. The timing of the algorithms was performed using the perf_counter() functionality available in Python's time library.

6.1 Experiment Parameters

In each generated experiment the algorithms are ran multiple times on the same data. Each instance with different parameter values in order to get a better impression of how the algorithm trade-offs work. Each algorithm is executed with up to a hundred different parameters and any algorithms using a lot of randomness are ran three times as often, using the same parameters, in order to find a more stable average performance. In more detail, the different algorithms are used as follows:

The random warm-up algorithm is ran three times to get a reasonable average, and has no parameters.

The small- and big-bucket algorithms are ran on parameters in the range from 1 to 30. This range is rather small because the small- and big-bucket algorithms converge at $d = \log N$, making this range large enough to cover any graph with less than a billion nodes. Since these algorithms only take integers as parameter, it only has 30 different iterations. Each parameter is only ran once, as these algorithms use very little randomization.

The static-dynamic algorithm is ran on parameters between 1 and 200, this range is quite wide, as the original paper [14] gives little guidance as far as picking proper values for this parameter goes. One hundred separate instances of the algorithm are ran on the same graph and update sequence, all with a different parameter within the given range. Each instance with a parameter value is also ran three times in order to obtain a more stable result, which is necessary because of the random warm-up part of this algorithm, which is being used as the dynamic black-box.

DC-Orient is only ran once, since it has no parameters that allow for a trade-off nor any randomness that makes it unpredictable.

The static simple algorithm is ran on the same parameter values as static-dynamic, since it is an improved variant of the same algorithm. It is also ran three times per parameter to obtain average results.

DC-Simple is ran on a hundred different probability parameters in the range 0.4 to 1. This range has been decided on because preliminary experiments have shown that parameters closer to 1 produce more interesting results than those closer to 0. Each parameter is ran three times for this algorithm as well, since a large component of this algorithm is the unpredictable random warm-up algorithm.

6.2 General Observations

Each of the following experiments will provide a number of graph pairs displaying the results obtained during that experiment. These graph pairs will always consist of a plot displaying the relation between average number of colors used and average recolors on the left side, and a plot displaying the relation between total time taken per instance of the algorithm and average recolors on the right side. Because the x-axis for both plots are identical, the two plots can be combined in order to obtain all three components of the algorithm output: number of colors, number of recolors and time taken.

The lines present in each plot represent the trade-off each algorithm can make. Each point a line passes through corresponds to one datapoint obtained by running the algorithm with a certain parameter. The arrow displayed on each line represents the direction of increasing parameters, meaning that the arrow points towards datapoints with higher parameter values, this allows us to intuitively understand the effect the parameter has on the trade-off, and whether increasing or decreasing its value is required to obtain the target results. Some of the algorithms are represented by a point rather than a line, this means the algorithm was not executed using multiple different parameters and no trade-off is therefore visible. In most cases this is due to the algorithms in question not having a trade-off parameter to begin with, but in the case of the Reddit dataset the long running times prevented more datapoints from being generated.

Because the importance of number of colors, number of recolors and running time can vary from application to application, it is difficult to define when an algorithm performs better than another algorithm. The plots therefore have to be read in a distinctive way. Since for all aspects, colors, recolors and running time, lower values are preferred, this means algorithms closer to the left-bottom corner are those that are most interesting. Algorithms that achieve similar results on one aspect as another algorithm, but outperform it on the others, are considered to produce better results, as they have a more efficient trade-off. We will therefore focus largely on whether lines or points lie left of or below other lines in order to compare them. Additionally, it is worth noting that the axis scales for each experiment are different, since each experiment uses a different graph and update sequence. The focus while comparing these various results will be the relative location between the algorithms within a plot, and how they compare to the ideal estimate for the average chromatic number C.

Most of the plots generated by the experiments look rather similar, we therefore first discuss the aspects of the results that are generally the same, before going into detail per experiment. We discuss the general observations per algorithm below. Figure 6.3 in Section 6.3 and Table A.9 in appendix A provide a visual and numerical representation of the most general experiment to support these observations.

Random Warm-Up This random warm-up algorithm, which has randomness as its main component, is built in such a way that it will only recolor at most one node during each update. Additionally, the relatively large color palette and random method of

assigning these colors makes the probability of a conflict occurring rather small when compared to the different algorithms, in which many of the nodes are assigned the same color if possible. This means that, in the random warm-up algorithm, often no nodes have to be recolored at all when an edge is inserted. This algorithm is therefore expected to produce extreme results skewed towards few recolors and short running time but high number of colors used. While we do see this extreme bias toward few recolors and short running time in the experiment results, one may notice that there are other algorithms that use even more colors than this one, even when requiring a similar number of recolors. This occurrence has very little to do with the performance of the random warm-up algorithm, however, and is likely due to the implementations of these other algorithms. More details about this occurrence can be found in the general observation sections of those algorithms. We will thus continue referring to the random warm-up result as one of the extremes, namely the one focusing on low number of recolors and short running time.

Small- and Big-Bucket Algorithms The small- and big-bucket algorithms cover a large range when it comes to their trade-off. The experiment plots show that, together, these two algorithms can reach from using very few colors but many recolors to the opposite case of using many colors but few recolors. All while also being competitive with the other algorithms, as the trade-off line of these two algorithms is curved towards the bottom-left corner. We can also indeed observe that as the parameters of the two algorithms increase, they produce more similar results until they converge in the middle. These two algorithms thus indeed seem to complement each other well, and allow for a wider trade-off than most other algorithms. The running time of the small- and big-bucket algorithms seems competitive with the other algorithms, as these results are very densely grouped together in most plots. As the number of recolors increases, the big- and small bucket algorithms, as well as the others, seem to increase in running time. This observation can be explained by the fact that the algorithms only have to perform non-trivial actions when nodes need to be recolored. If an update occurs in which no nodes have to be recolored, the algorithm does not have to perform a significant amount of work and the running time will thus increase based on number of recolors. Additionally, it is worth noting that the small-bucket algorithm produces results with considerably more colors used than the random warm-up algorithm only when its parameter is set to 1. It is unsurprising that this algorithm produces such extreme results with this parameter, since a parameter value of 1 will cause the bucket structure to consist of single level with a high number of buckets that can only fit one node, each with a distinct color palette. This result therefore simply indicates that the small-bucket algorithm was not meant to be used with parameter values that low.

Static-Dynamic Algorithm As expected, the experiment plots show that, even in the same range of recolors, the static-dynamic algorithm requires many more colors than all other algorithms (with the exception of the small-bucket algorithm with parameter 1). This result is explained by the fact that this algorithm uses two different colorings, the

static coloring and the dynamic coloring, which it combines by creating color tuples (c1, c2). This process practically multiplies the number of colors used in the static coloring with those used in the dynamic coloring, creating an unnecessarily large color palette. An interesting observation is that, if the parameter used is very small, the static-dynamic algorithm produces results closer to the competitive area of the other algorithms and in the direction of DC-Orient. This is because at very low parameter values the static-dynamic algorithm executes a full static recolor almost every update. Since DC-Orient simulates the colorings the greedy static coloring algorithm would produce (which is the static black-box used in static-dynamic) these two algorithms end up producing more similar results. Similarly at very high parameter values, static-dynamic seems to produce results more similar to the random warm-up algorithm. Once again this effect is explained by the fact that at very high parameter values a static reset almost never occurs and every step is simply a dynamic black-box step, and thus in this case a random warm-up step. The interesting part of this behaviour is that while many more algorithms within the plots display such a balance between the random warm-up and DC-Orient, this static-dynamic line does not curve to the bottom-left, but rather upwards. This peak reinforces the belief that the multiplication of the static and dynamic colorings is the cause of the high number of colors used, since this multiplication would reach its highest value when the static and dynamic parts of the algorithm have a similar importance, and thus a similar number of colors. The static-dynamic algorithm does seem to be competitive when it comes to running time, but is not significantly faster to compensate for the high number of colors used.

DC-Orient Almost all left-hand side plots show DC-Orient in the far bottom-right corner, making this algorithm the second extreme option. Where the random warm-up focuses on low number of recolors and has short running time, we can see that DC-Orient has a focus on low number of colors used and has the highest running time of all the algorithms. These results are as expected, since the DC-Orient algorithm was not designed with a focus on low number of recolors. While the running time and high number of recolors make this algorithm viable only if the quality of the colorings is of utmost importance, it does provide us with a second anchor point, together with the random warm-up, for our two new algorithms.

Static-Simple Static-simple was designed to be an improvement over static-dynamic. Where static-dynamic combined the random warm-up algorithm and DC-Orient by using them as black boxes and later combining their separate colorings, static-simple instead uses only a single coloring. This simple change in the way these algorithms are combined removes the blow-up of colors used that was present in static-dynamic and, as can be seen in the experiment plots, reduces the number of colors significantly while retaining the same range when it comes to recolors. The running time, while differing somewhat, is still competitive with static-dynamic and the other algorithms and overall it thus seems that this improved variant of static-dynamic has succeeded in its goal. The relatively small range of recolors that was covered by static-dynamic, especially when compared

to the small- and big-bucket algorithms, has not been increased however, meaning that while the trade-off is much more efficient, it is not as versatile as certain other options.

DC-Simple Finally, DC-Simple, created to be the simplest combination of the random warm-up algorithm and DC-Orient is shown to have succeeded in its purpose. Not only does its trade-off line move accurately towards both of these algorithm results, it does so with a bottom-left curve that is lower than the other algorithms considered so far. This means that the trade-off between colors and recolors offered by this algorithm is the most efficient one so far, making this the ideal option when both colors and recolors are important in the application in question. This efficient trade-off comes at a cost, however, since the running time aspect of this algorithm is less than desirable for most parameter values. While the running time of DC-Simple does considerably improve upon that of DC-Orient, even for relatively low parameter values, it is only competitive with the other algorithms at high ones. This high running time is something to consider when deciding on an algorithm for a specific application. Additonally, peaks in the trendline for running time can be observed in some of the plots for DC-Simple. These peaks are likely not due to the parameter values of the algorithm, but rather to the way the experiments were executed. Because of the long running time for this algorithm, the collection of the datapoints required to generate the plots has taken a considerable amount of time. During this timeframe the device used to run the experiments was occasionally also used to perform other, short-term, tasks. These background processes have likely caused the computer to slow down its processing speed, causing an occasional peak in the timeline of this algorithm.

With these general observations about the different algorithms clarified, the rest of the experiments will focus on the differences in the results that occur when running the algorithms on different types of graphs and update sequences. These experiments will allow insight into different situations in which certain algorithms might have an advantage due to how they work internally.

6.3 Small vs. Large graphs

The first and simplest of the experiments is the one related to the size of the generated graph. Various algorithms, like the bucket-based and the static-dynamic algorithms, rely on the number of nodes in a graph when creating its underlying data structure. The small-bucket algorithm will create more buckets per level as the number of nodes in a graph increases and the static-dynamic algorithm will have more levels of update segments. We hypothesise that these algorithms will produce better results on larger graphs, since they will get the opportunity to use their data structures to the fullest.

In order to test this hypothesis, we run all algorithms on graphs of different sizes. We do this by generating random graphs with increasing amount of nodes and edges and take the update sequence to be a one by one addition of the graph's edges in a random order. We pick this randomized method to reduce bias as much as possible.

The results for increasing number of nodes can be found in Figures 6.2, 6.3 and 6.4 and the those for increasing number of edges in Figures 6.5, 6.6 and 6.7. In these figures, two plots are shown, the left of which displays the trade-off between colors used and recolors, and the right of which displays the running time as related to the number of recolors. By looking at these graphs side-by-side the relationship between number of colors used and running time can also be deduced. Additionally, a zoomed-in version of the time plot from Figure 6.4 is provided in Figure 6.1. This zoomed variant allows us to see how the running times compare of the algorithms that are grouped close together in most other plots. The experiment with a large graph has been chosen to zoom into, as this will provide us with more stable, and thus more representative values, as compared to smaller graphs.

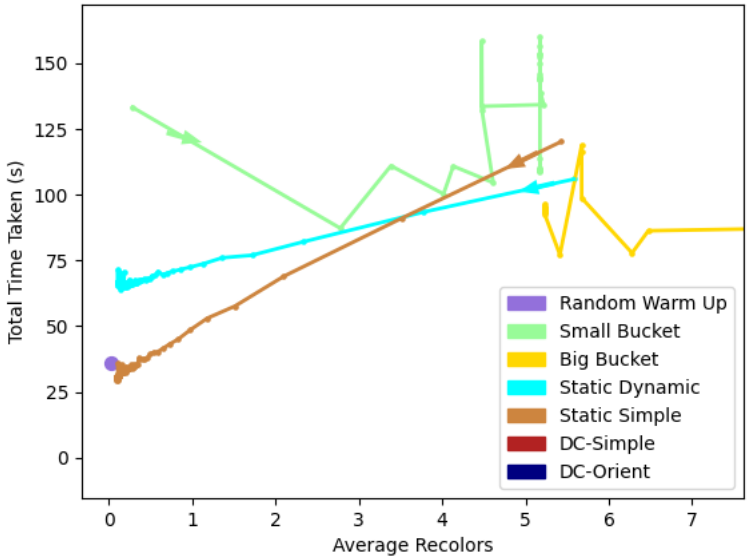


Figure 6.1: Zoomed-in plot of the running time from Figure 6.4.

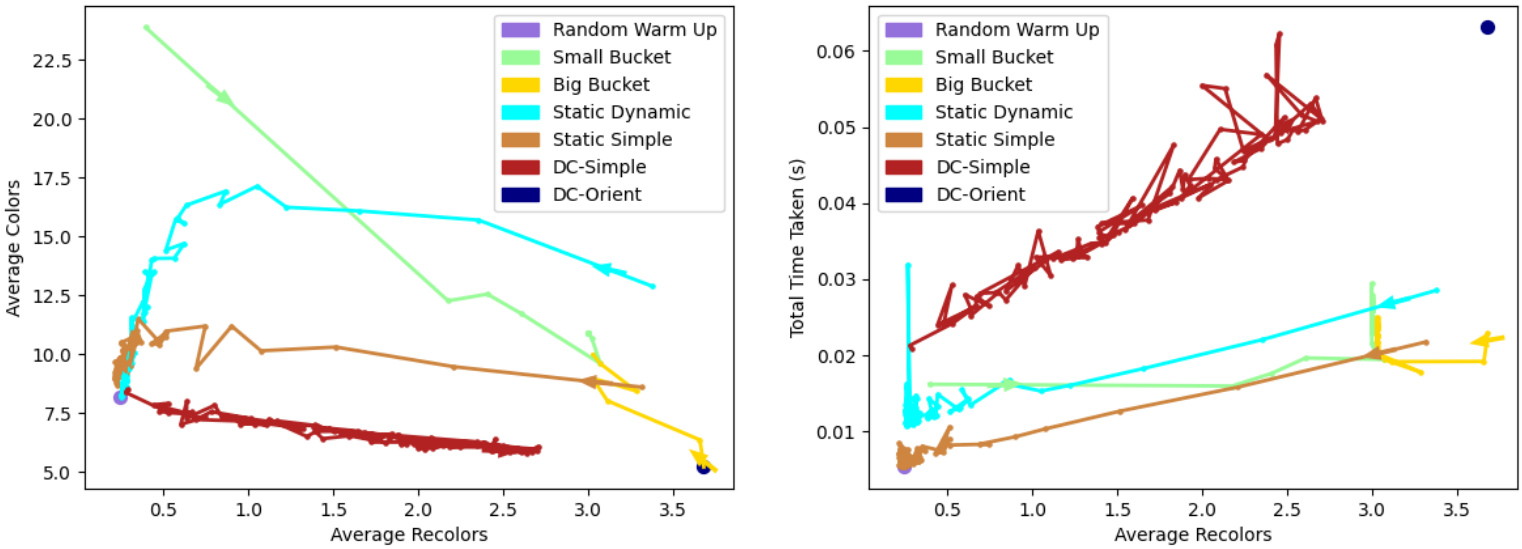


Figure 6.2: Results for a small number of nodes with random edges and random update sequence. 30 nodes, 217 edges and estimated average C = 5.21.

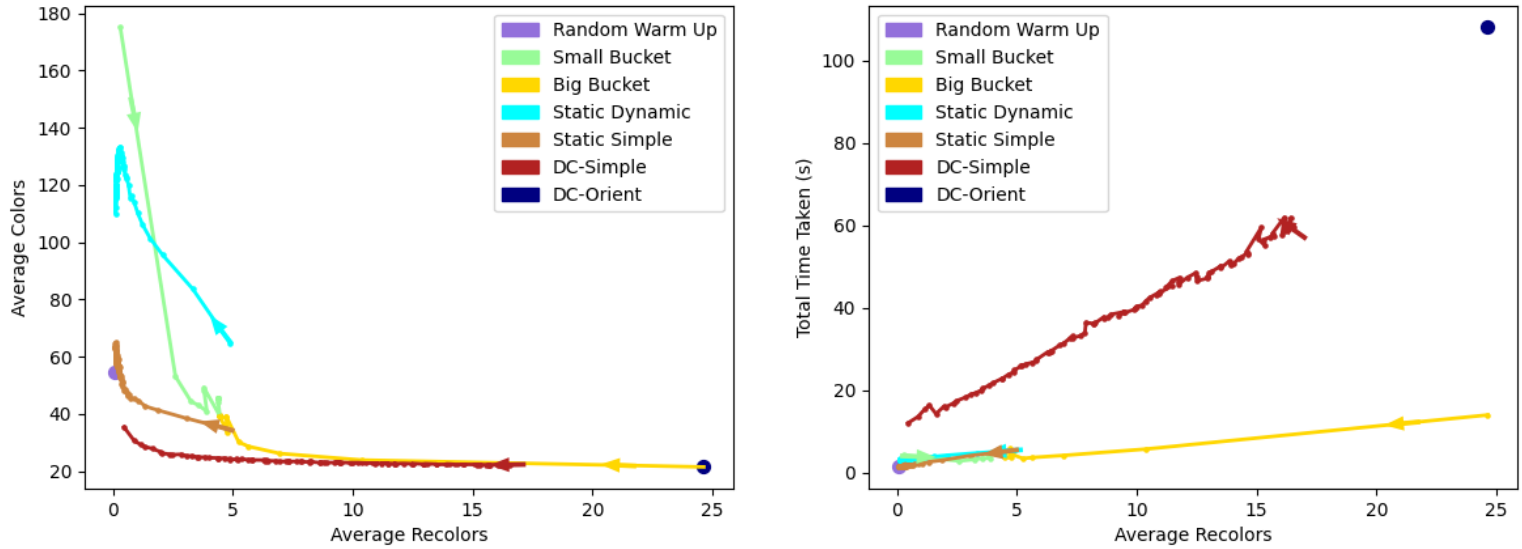


Figure 6.3: Results for a medium number of nodes with random edges and random update sequence. 200 nodes, 11940 edges and estimated average C = 22.

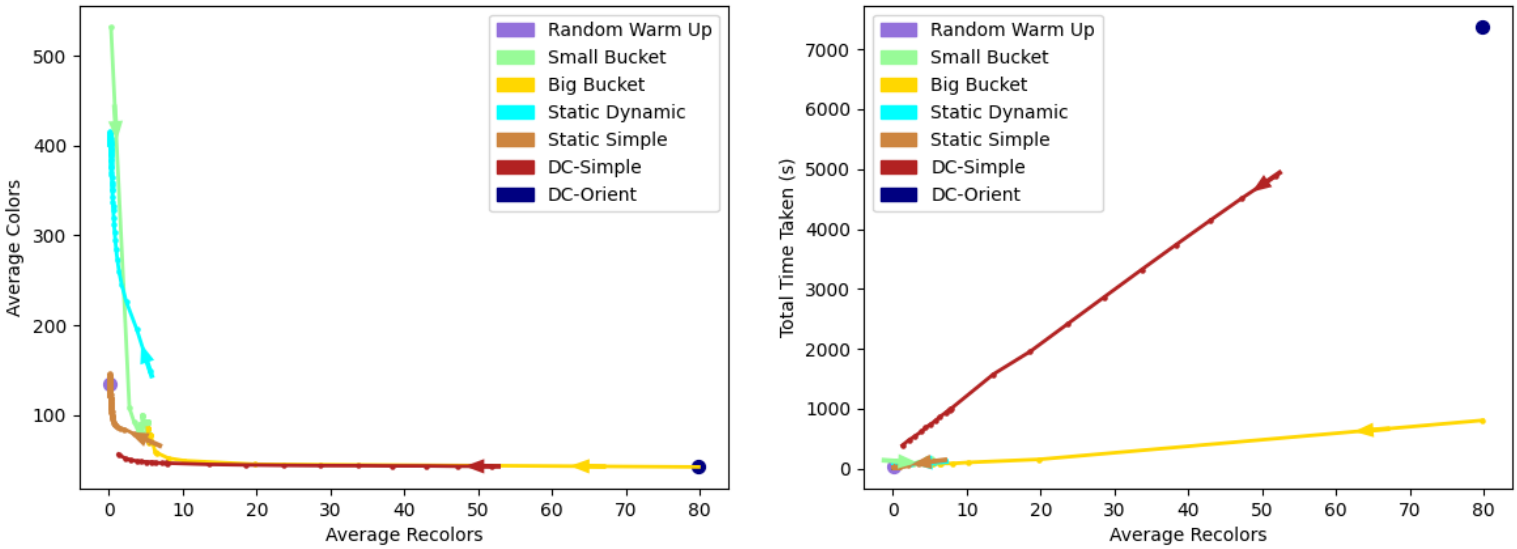


Figure 6.4: Results for a large number of nodes with random edges and random update sequence. 600 nodes, 89850 edges and estimated average C = 42.

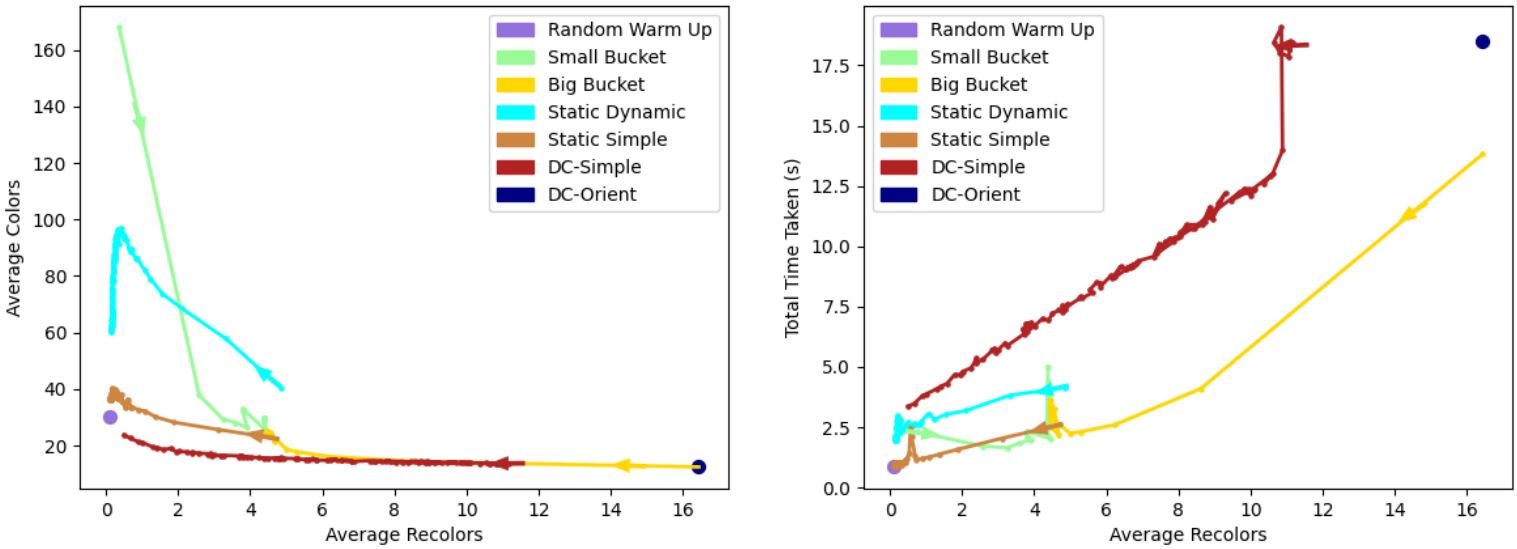


Figure 6.5: Results for a small number of random edges and random update sequence. 200 nodes, 5970 edges and estimated average C = 12.

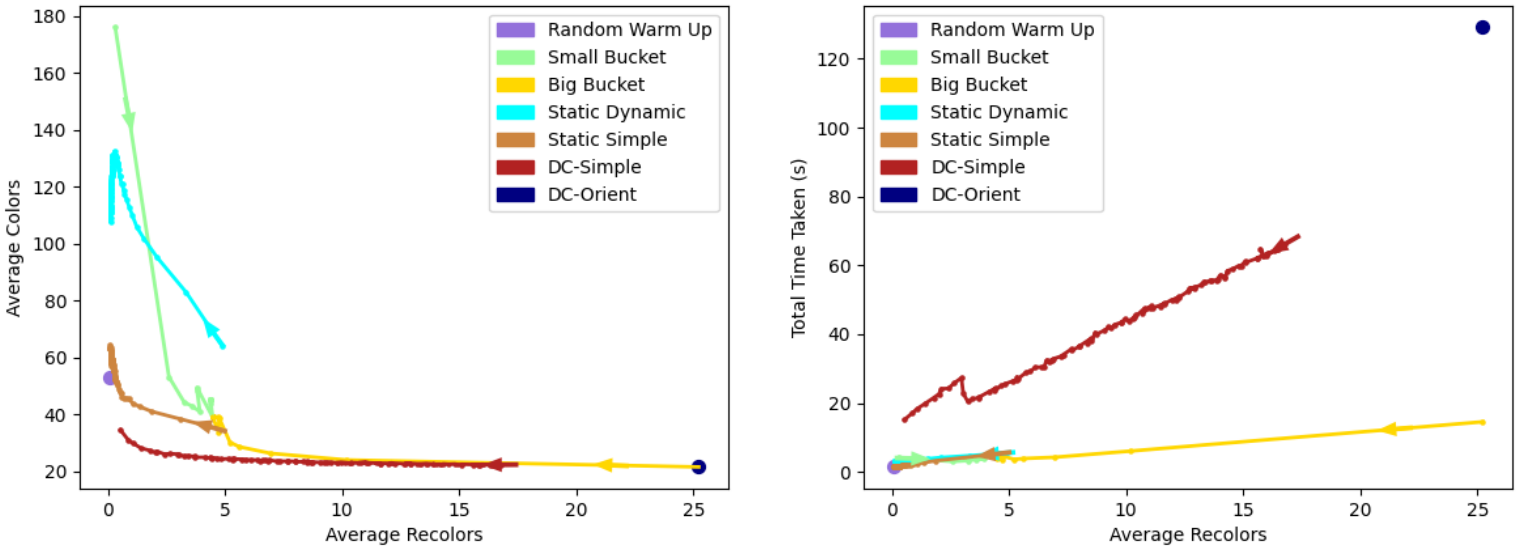


Figure 6.6: Results for a medium number of random edges and random update sequence. 200 nodes, 11940 edges and estimated average C = 21.

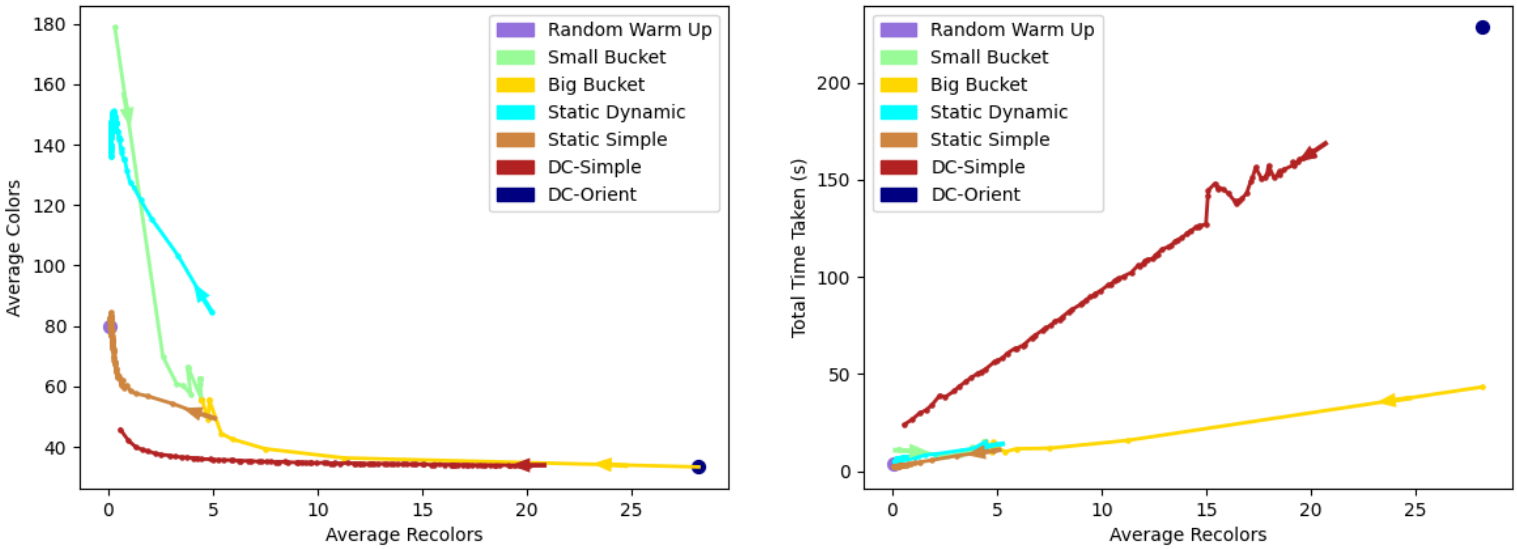


Figure 6.7: Results for a large number of random edges and random update sequence. 200 nodes, 17910 edges and estimated average C = 33.

In the results for increasing number of nodes, it is clearly visible that at a small number of nodes, three different algorithms eventually converge at the random warm-up results. The static-dynamic, static-simple and DC-Simple algorithms do indeed all make use of this random warm-up as a subroutine and an extreme value for their parameters can cause their results to become very similar. As the number of nodes increases, however, we see that only static-simple and DC-Simple remain in the same range as the random warm-up, whereas the number of colors used in static-dynamic are immediately too high to be competitive. For static-dynamic we thus conclude that the multiplication occurring when combining its two colorings outweighs the advantage obtained from using a reset step during the update sequence, making it unsuitable for large graphs. The two algorithms that remain in a similar range as the random warm-up, together with this random warm-up itself do seem to remain quite competitive even as the number of nodes increases, although the number of colors used by the random warm-up increases at a much faster rate than our estimate for C. This acceleration could indicate that the random warm-up and the two combination algorithms, with high parameters, may not scale well on even larger graphs, potentially due to the lack of reset steps in the algorithms, resulting in more suboptimal colors being left behind in the coloring, or potentially due to the average degree being higher in larger graphs, thus increasing the palette size of these algorithms. The small- and big-bucket algorithms display the opposite effect: in the small example, the combined line of these algorithms is almost straight from the top-left to the bottom-right corner, only a small part of this range is interesting, as the other algorithms perform objectively better for most of the range. As the number of nodes increases in the medium and large example, however, we find that this combined line curves to the bottom-left in an increasingly strong manner, making the algorithms competitive on a much larger part of the covered range. Additionally, the big-bucket algorithm is the only one that achieves similar number of colors used as DC-Simple and DC-Orient, while requiring a lot less time to do so. The small- and big-bucket algorithms thus show themselves to be scalable, likely due to their reset behaviour. Finally, DC-Orient consistently performs well when it comes to number of colors used, but as the graphs become larger starts falling behind in number of recolors and running time quickly. This also explains why the running time line of DC-Simple seems to become steeper as the graphs get larger: it traces a line between the random warm-up and DC-Orient, and DC-Orient increases its running time at a much faster rate than the random warm-up does.

The increasing density experiment shows us similar occurrences, albeit in a much more minimal manner. The static-dynamic number of colors still blow up, but the random warm-up number of colors used increases at a similar rate to the estimate of C, and the number of recolors for DC-Orient barely changes at all between the medium and large density tests. Additionally we can observe that the strength of the curve for the small- and big-bucket algorithms remains largely the same over the course of the different experiments, indicating that the density of the graph has little to no effect on its performance.

Finally, in the zoomed-in plot of Figure 6.1 we can see that the random warm-up

does seem to be the fastest option in general, allowing for some random variation when comparing it to the static-simple line. Apart from this result, however, it seems none of the algorithms consistently perform the quickest. The static-dynamic and static-simple algorithms are both almost as fast as the random warm-up when their parameters are high, but as their parameters decrease become on-par with the small- and big-bucket algorithms, at which point they would increase at a faster rate if not for the fact that the parameter range ends here. The small- and big-bucket algorithm seems the most consistent in its running time at this small scale, but Figure 6.4 shows that there is some increase in running time at the large scale, albeit much lighter than that of DC-Simple.

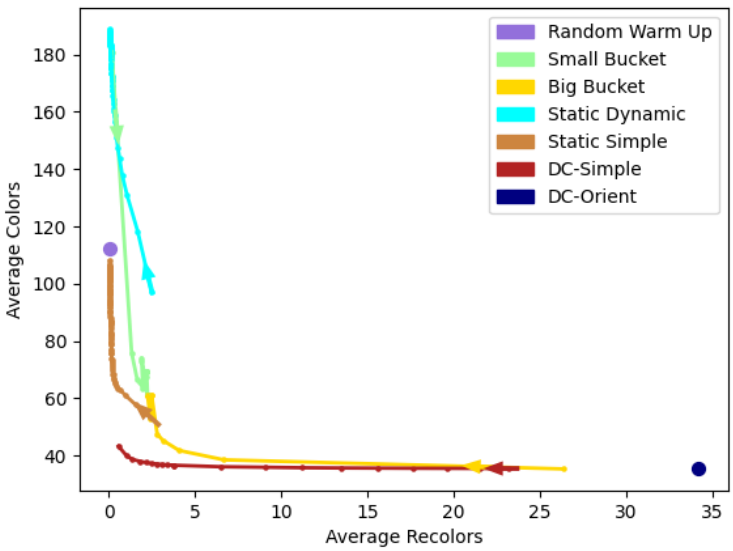
6.4 Constant Update Stream

In this experiment we generate a stream of updates including both edge additions and removals, such that many updates are executed without changing the properties like size and density of the graph much. We generate two different update streams of a hundred thousand updates, one in which random edges are removed to maintain the same size and the other in which older edges are removed with a higher probability. The starting point of both update sequences is a random graph of a medium size, with 200 nodes and 11940 edges.

For the update sequence in which older edges are more likely to be removed, which we call the decaying update stream, initially each edge in the graph has the same probability to be removed once an edge removal step occurs. To signify this, all edges are assigned a weight of 1. With each update that is performed, all edges in the graph increase their weight by 1. As such, older edges will have a higher weight. When a new edge is added to the graph, its weight is initialized at 1, making it much less likely to be removed during an edge removal step than those already present in the graph for longer. When an edge removal step needs to occur in order to keep the number of edges in the graph stable, a weighted random selection is done, selecting one edge from the set of edges currently present in the graph, based on their current weight.

This experiment is meant to give an insight in how well the algorithms deal with extended use, and whether or not they will start performing worse over time if they were to be implemented in a real-life application. We expect algorithms that include full resets, such as the small- and big-bucket algorithms to not be negatively affected in this situation, but more randomized algorithms such as the random warm-up or the dynamic part of static-dynamic might cause the number of colors used to increase significantly in such an update stream.

The results are provided in Figures 6.8 and 6.9.



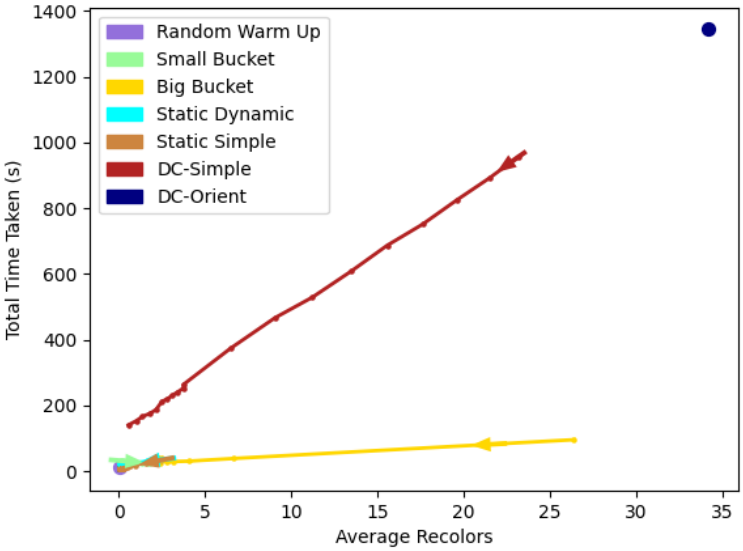
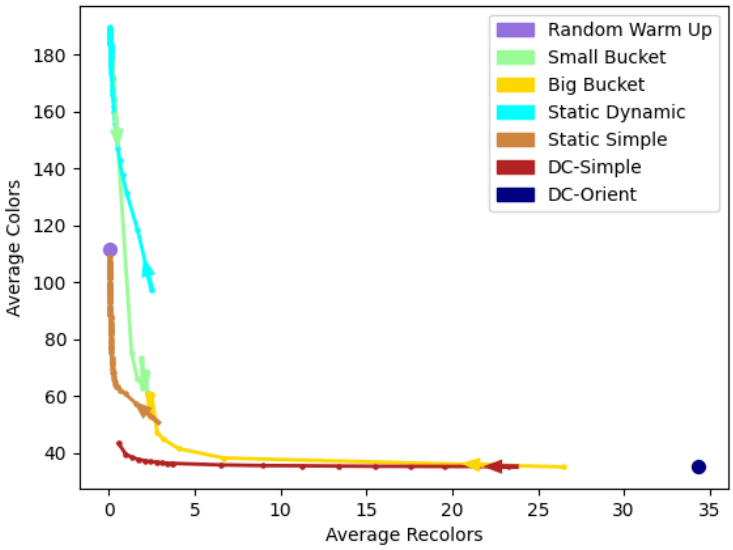


Figure 6.8: Results for a random stream of 100.000 updates, 200 nodes and estimated average C = 35.



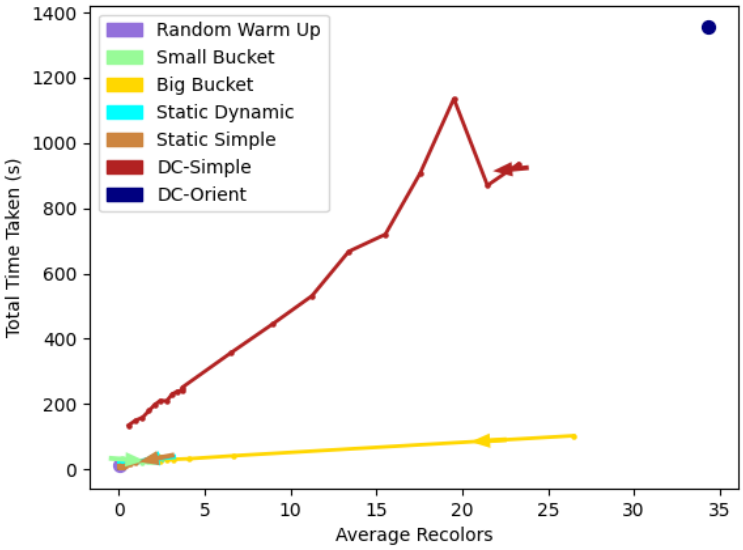


Figure 6.9: Results for a decaying stream of 100.000 updates, 200 nodes and estimated average C = 35.

From these results it is clear, when comparing them to the results from Section 6.3, that indeed algorithms with reset functionality perform better on such long update streams than the more randomized algorithms do. The random warm-up point lies much higher as compared to the other algorithms, for example, whereas the small- and big-bucket algorithms remain in the same range. The static-dynamic algorithm performs relatively well, as it is quite close to the random warm-up and static-simple. It seems the reset functionality of its static component may be countering the increase of colors used caused by the otherwise detrimental multiplication of static and dynamic colors. Surprisingly DC-Simple does not produce significantly worse results when executing 100.000 updates, even when nodes are colored randomly with a very high probability parameter. This would indicate that 'incorrect' colors caused by taking random coloring steps are not often left behind in the coloring and are relatively quickly overwritten by a DC-Orient CAN step. This result, combined with the fact that at high parameter values the running time of DC-Simple is almost competitive with the other algorithms, makes it a strong contender in the case of long update streams. The difference between a random stream or a decaying stream seems negligible from these results, as the only considerable difference is the peak in the DC-Simple time line, which is likely caused by external factors as explained in Section 6.2.

6.5 Degree Variation

In this section we explore the effect degree variation has on the different algorithms. We run the algorithms on graphs with similar sizes and randomly ordered updates, but vary the nodes these updates focus on. For the first experiment, random edges are selected, making the degree distribution in the graph mostly fair, secondly a light and heavily skewed degree distribution are generated using node priorities, in order to see the effect an uneven degree distribution has on the algorithms. Finally, an experiment is ran in which the degree is mostly fairly distributed, apart from a single node with a very high degree. This difference in the maximum degree of the graph could have an effect on the algorithms that select a color from a palette based on the degree of a node, such as the random warm-up.

For the light and heavily skewed experiments, two graphs are generated in which edges are unevenly distributed. To achieve this, we use a process that makes edges adjacent to some nodes more likely to occur in the initial graph. A process which we call node prioritization. This prioritization is achieved by assigning each node a priority between 0 and 1 during the graph generation phase. Whenever an edge is considered for being part of the initial graph the edge priority is obtained by combining the node priorities of the two adjacent nodes. This edge priority indicates the probability it is indeed added to the graph, decided by a weighted coin toss. If the result of this coin toss is negative, a new edge, which could potentially be the same one, is selected for consideration until the target number of edges is obtained. The distribution of these edge priorities thus influences how skewed the degrees in the initial graph become. If the edge priorities are fairly similar, only a light skew occurs, whereas if edge priorities

differ more, a heavier skew will occur in the resulting graph. To achieve both these options we combine the node priorities into edge priorities in two different ways. In order to get a light skew we combine node priorities into edge priorities by taking their average, whereas in order to obtain a heavy skew we combine node priorities by squaring them both and multiplying them together. This square and multiplication enlarges the difference between the original node priorities and thus cause the edge priorities to be further apart. As such, we obtain both a graph with lightly skewed degree distribution and one with heavily skewed degree distribution.

The results of this experiment can be found in Figures 6.10, 6.11, 6.12 and 6.13

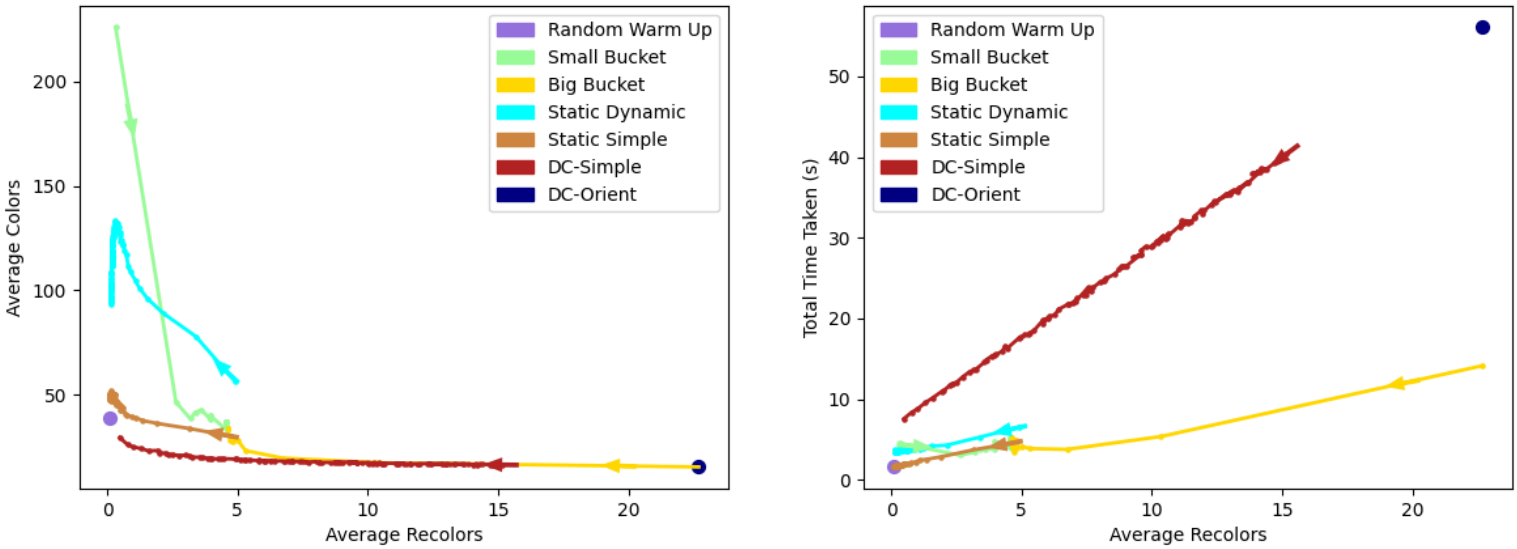


Figure 6.10: Results for a fair distribution of degrees. 265 nodes, 10623 edges and estimated average C = 15.

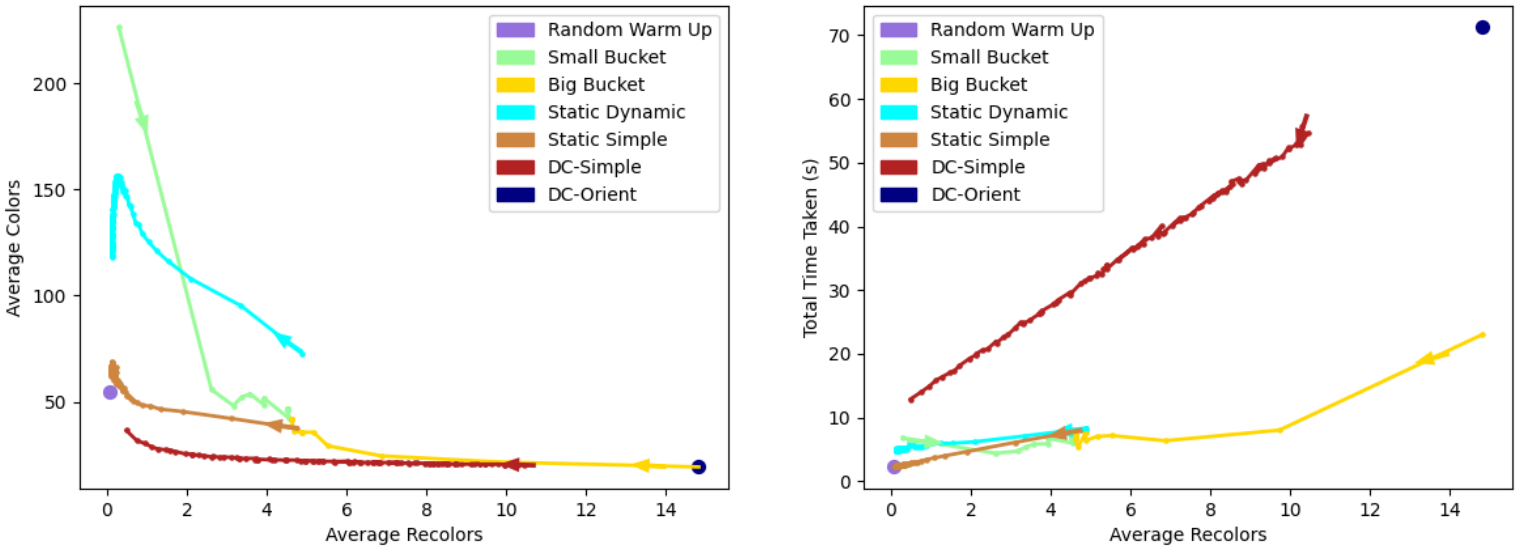
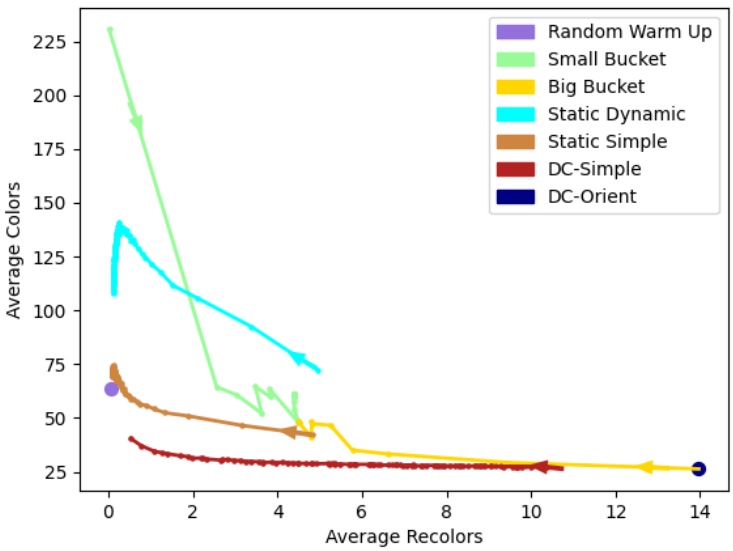


Figure 6.11: Results for a lightly skewed distribution of degrees. 259 nodes, 14852 edges and estimated average C = 19.



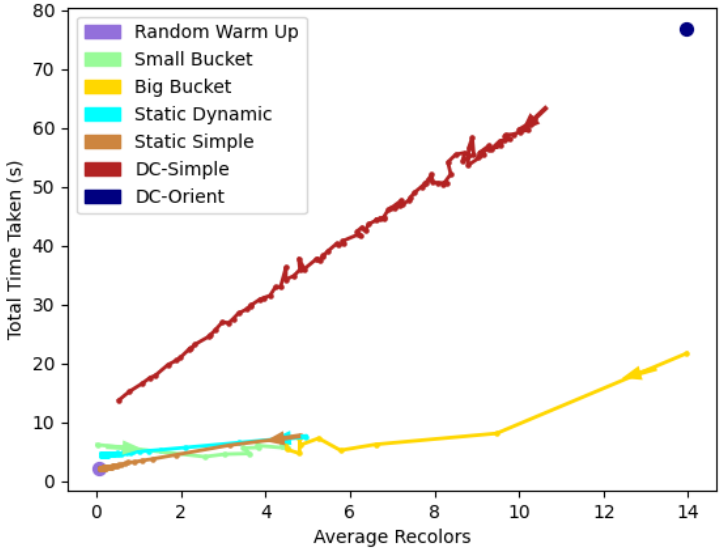
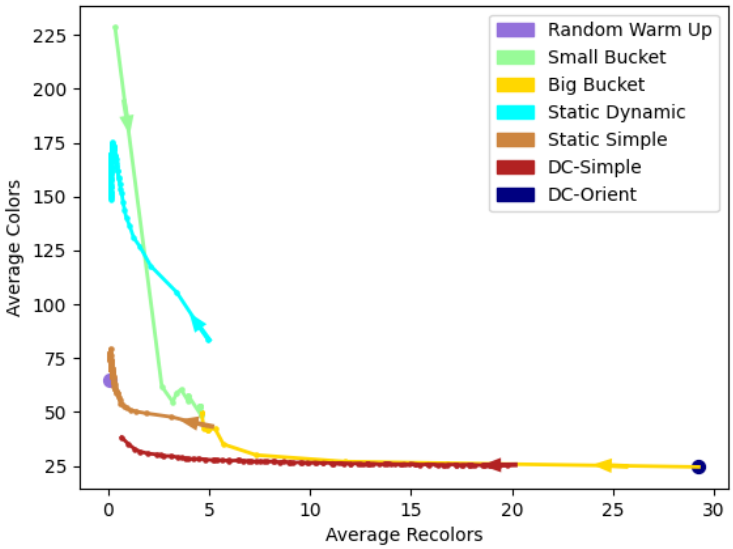


Figure 6.12: Results for a heavily skewed distribution of degrees. 250 nodes, 13769 edges and estimated average C = 26.



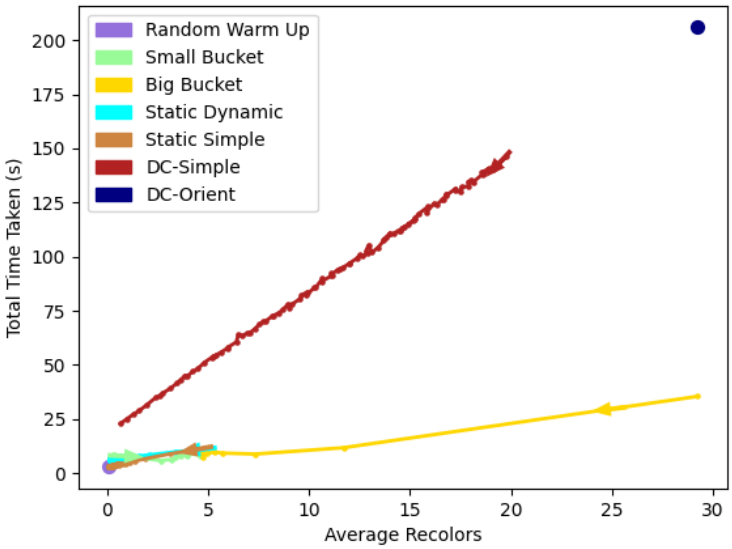


Figure 6.13: Results for a mostly fair degree distribution with a single high degree. 264 nodes, 19167 edges, maximum degree of 263 and estimated average C = 25.

From these results we can conclude that almost all algorithms have very similar performance as in other experiments, regardless of the distribution of degrees. The only point of interest here is that DC-Orient, and to a lesser extent therefore also DC-Simple, seem to require fewer recolors as the distribution of degrees becomes skewed, an effect which causes the other lines in the plot to appear more spread-out horizontally. We thus conclude DC-Orient is the only algorithm performing more efficiently on a graph with skewed degree distribution. This aspect of DC-Orient makes it a more viable option in certain biased situations, rather than only being an algorithm focused on number of colors. This effect does not seem present when only a single node of high degree is added to the graph, nor does this addition seem to significantly affect the other algorithms.

6.6 Update Spread

This section focuses not on the edges added during the update sequence, but on the order in which these updates are executed. We hypothesise that this order will make a large difference in algorithms such as static-dynamic, since it only statically recolors nodes with a high recent degree, obtained by adding many edges to a node in relatively few updates. The three methods used to generate this update order are: randomly, prioritized based on an expanding breadth-first-search principle, where all edges are likely to be part of a single connected component, and lastly prioritized on node, where updates adjacent to a node with high priority are more likely to occur early in the update sequence.

Note that at the point in the experiment setup process at which the update order is decided upon, the edges which will make up the edge addition updates are already determined, as these are the edges in the already generated random graph. The variety of update spread is only due to the order in which these edges are added. As such, the random update order is simply obtained by taking the already selected edges and shuffling them at random.

The expanding order is achieved by initially assigning each node a weight value of 1. The first edge in the update sequence is then selected by a weighted random sample using calculated edge weights, which are obtained by summing the node weights of the edge's endpoints. The nodes adjacent to the selected edge increase their weight by 1 before selecting the edge to be added next in the update sequence. This makes edges adjacent to a node that already has an edge in the update sequence more likely to be added sooner, but does not have a strong enough effect as to focus edge additions around a specific node. This process continues until all edge addition updates are ordered and provides us with an order reminiscent of a breadth-first-search approach.

The node prioritized update sequence is generated by assigning each node a priority between 1 and 1000, after which these priorities are taken to the power of some parameter x. This parameter can be adjusted to increase or decrease the strength of the prioritization. As found in preliminary experiments, however, this approach requires the parameter to be quite large, only clearly showing the intended effect of focusing mostly on one node at a time when using a value of 100. The result of this computation is

thus that the priorities of the nodes are very spread out. After this computation each node has a set priority and edges are selected as follows: first a node is selected using a weighted random sample using the node priorities. From the set of available edge additions, a list is created of edges that are adjacent to this selected node, and from this list one edge is selected at random. This edge is put first in the update order and the whole process is repeated for the next position. If a node has no more adjacent edges in the available edge update set, it is removed from the random sample pool. This manner of ordering the edges is very likely to first add all available edges adjacent to the node with the highest priority and after that continue with adding all available edges that are adjacent to the node with the second highest priority, and so on. The ordering resulting from this method is thus grouped strongly.

The results for this experiment can be found in Figures 6.14, 6.15 and 6.16.

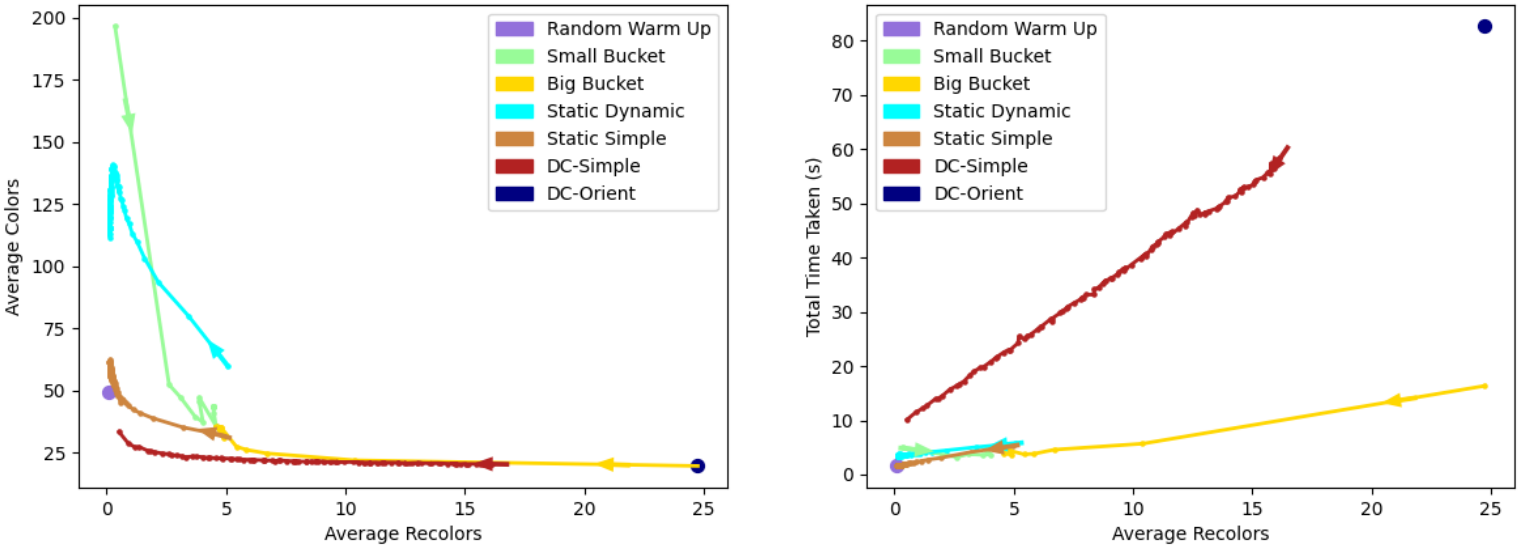
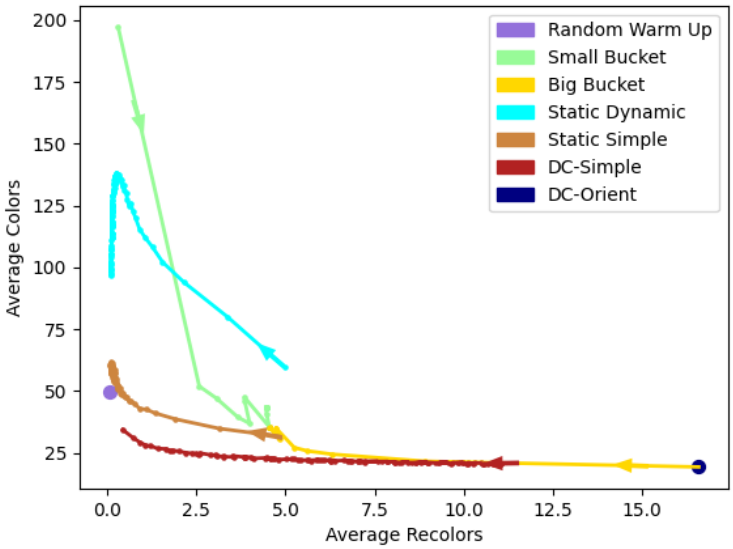


Figure 6.14: Results for a randomly ordered update sequence. 228 nodes, 12440 edges and estimated average C = 20.



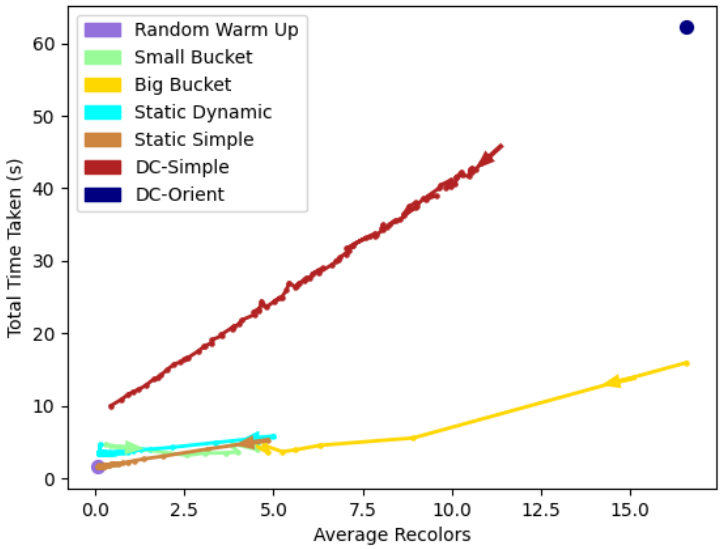
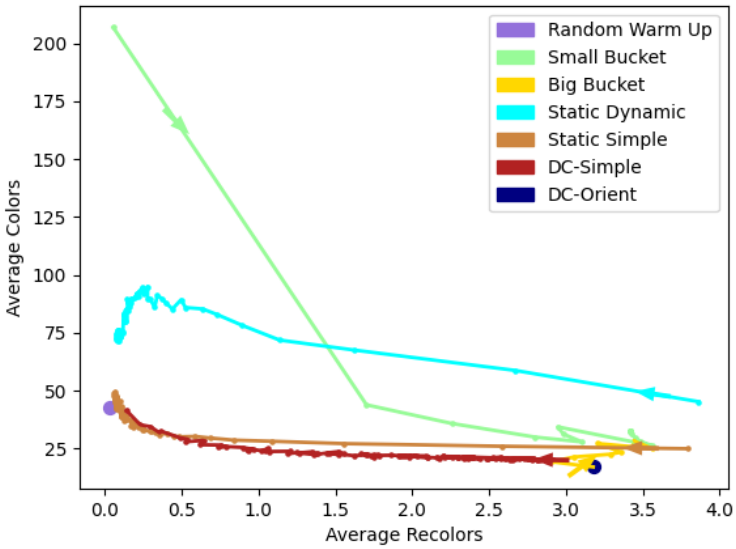


Figure 6.15: Results for an expanding update sequence. 228 nodes, 12440 edges and estimated average C = 19.



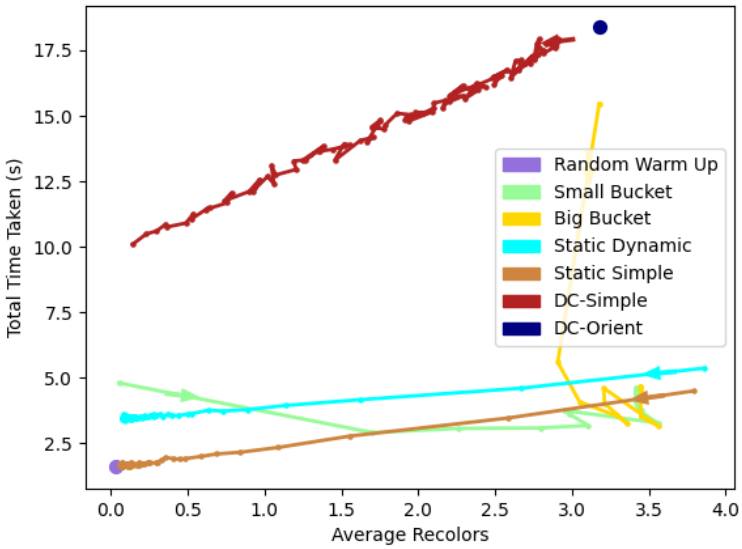


Figure 6.16: Results for an update sequence with strong node prioritization. 228 nodes, 12440 edges and estimated average C = 17.

Indeed we find that the static-dynamic algorithm is once again lowered in number of colors used to become closer to the results of the random warm-up, especially in the case of the node prioritized experiment. Additionally we once again see DC-Orient profit from a skewed update set. In this case it achieves fewer recolors in the expanding update sequence, while still obtaining the optimal number of total colors used. The same, but much more extreme effect can be observed in the results for the node prioritized update sequence, where the average number of recolors for DC-Orient drops as low as the convergence point of the small- and big-bucket algorithms. This decrease in recolors also allows the running time of DC-Orient to be almost equal to the other, usually faster, algorithms. This surprisingly effective behaviour of DC-Orient, especially combined with the results from the degree distribution experiment in Section 6.5, make it a much more versatile option than initially expected. The DC-Simple algorithm can be seen to profit to a lesser extend from these same skewed update sequences, as it only uses a DC-Orient step occasionally, depending on the chosen parameter. This surprising location of the DC-Orient datapoint also effects the big-bucket line, as the big-bucket algorithm with a very low parameter often uses a reset step, which has been implemented using the static greedy coloring algorithm that DC-Orient aims to simulate. It is therefore also visible in these plots that the big-bucket algorithm line moves towards the position of DC-Orient, making for some strange curves, as this is not where the DC-Orient datapoint would normally lie.

6.7 Reddit Dataset

For the final experiment, we use a real-life dataset representing hyperlinks between different 'subreddits' on the social media platform Reddit. This dataset originates from a Stanford University research paper by Kumar et al. [17]. Some preprocessing was done to make this dataset applicable to our work. The directed edges in the dataset were interpreted as undirected ones and any duplicate edges caused by this process were removed. Remaining are 35776 nodes and 124330 edges, which the greedy static coloring algorithm can color using only 34 colors. The graph is thus large, but not very dense. In this experiment we hope to see which algorithms work best in a real-life example, rather than generated graphs with specific properties.

The results of this experiment can be found both in Table 6.1 and Figure 6.17.

Table 6.1: The results of running the algorithms on a real-life Reddit dataset.

35776 Nodes; 124330 Edges; C = 23.05	Average nr. of Recolors	Average nr. of Colors	Time Taken (s)
Random Warm-Up	0.11	78.3	3321.593
Small-Bucket algorithm (d = 5)	3.93	137.31	3494.585
Big-Bucket algorithm (d = 5)	5.39	51.1	3167.058
Static-Dynamic algorithm (l = 10)	1.24	283.54	7779.184
Static-Dynamic algorithm (l = 100)	0.22	343.64	7375.252
DC-Orient	2.48	23.05	7444.115
Static-Simple algorithm (l = 10)	1.13	94.65	2862.408
Static-Simple algorithm (l = 100)	0.2	93.35	2904.748
DC-Simple (p = 0.8)	24.21	37.16	15795.713
DC-Simple (p = 0.998)	2.9	61.8	6857.528

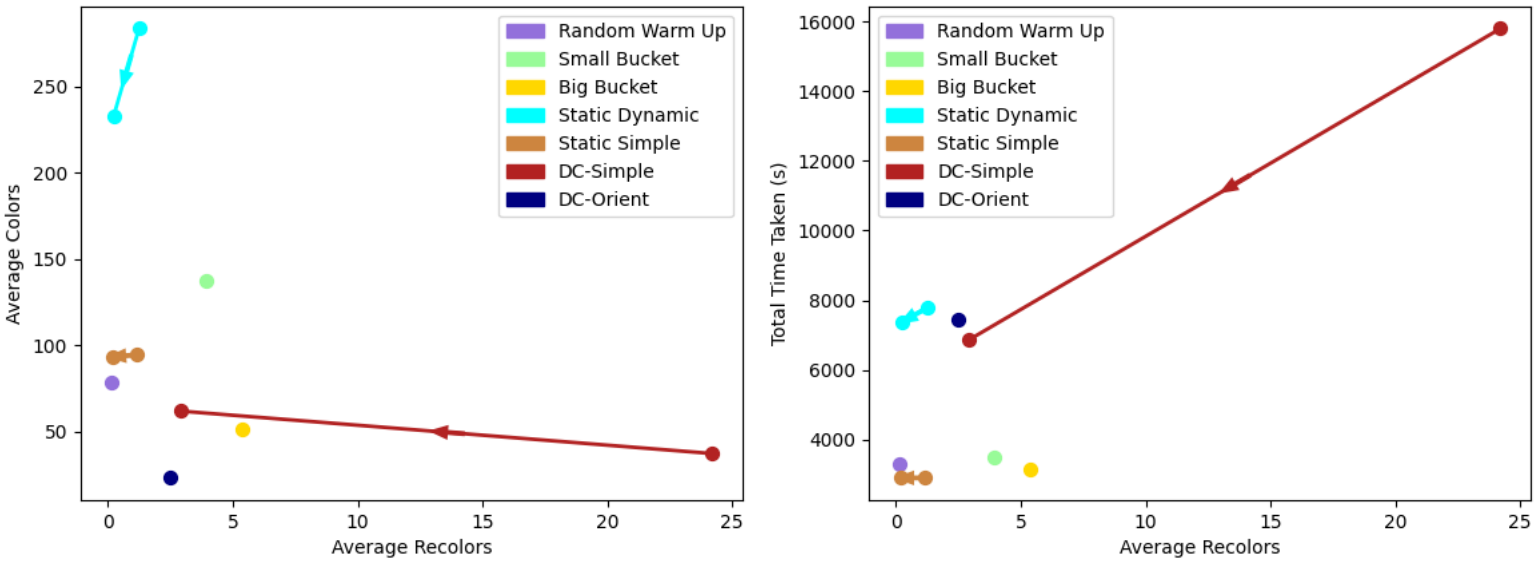


Figure 6.17: Visual plot of the results of the real-life Reddit dataset, based on the values of Table 6.1

We find that most algorithms behave as expected. The random warm-up uses relatively many colors but has a very low number of recolors and running time. The small- and big- bucket algorithms do not provide the best results but do achieve a fairly efficient trade-off. Static-dynamic uses a total number of colors that is much higher than all other algorithms and static-simple simply improves upon static-dynamic in all aspects. What is surprising about these results is how well DC-Orient manages to perform on a large real-life dataset. Whereas previously DC-Orient has usually been the slowest algorithm

with by far the most recolors, in this case the number of recolors is quite competitive with the other algorithms even when it has the lowest total number of colors used. Additionally the running time for DC-Orient is on par with static-dynamic and even lower than the DC-Simple algorithm for low parameter values, this is unlike what could be observed with the generated graphs. In this case DC-Simple is not able to outperform DC-Orient, although the final instance with $p = 0.998$ comes close. The efficiency of DC-Orient on this real-life dataset could be due to an inherent skew in either the edges or order of the updates in the dataset. As has been shown in Sections 6.5 and 6.6, DC-Orient performs better on datasets with some aspect that is not evenly distributed. The authors of the paper from which the dataset originates [7] do mention that many 'conflicts' between subreddits are caused by only 1% of the existing subreddits. These concentrated conflicts could be the cause of many cross-referencing hyperlinks between such 'aggressive' subreddits and make it likely for this dataset to indeed contain an unfair distribution of edges. Since the effect observed here is much stronger than seen during the generated experiments, including the experiment from Section 6.6, which included a very strong skew already, it seems that this real-life dataset may contain biases that are much stronger than anticipated.

Chapter 7

Conclusion

From the results obtained in Chapter 6 a number of conclusions can be drawn. These conclusions are grouped per algorithm and presented in this chapter.

Random Warm-Up The random warm-up algorithm has shown to only be a good option when dealing with a rather small graph with a relatively short lifetime. If the graph grows too large or undergoes a large number of updates, the randomness of this approach will cause the total number of colors to be higher than necessary. Even so, it is known to theoretically be bounded by the maximum degree $\Delta + 1$, and if the number of recolors or running time is of most importance, there is no other algorithm that achieves a lower number of recolors or faster running time than this one.

Small- and Big-Bucket Algorithms These algorithms turned out to be the baseline, or middle of the road, when it comes to the trade-off between recolors and total number of colors used. The two complementary algorithms cover the entire range of the trade-off by using different parameters, as is evident from the figures in Chapter 6, and perform competitively in all except the most extreme cases. While not the most efficient solution for every case, these algorithms are a reasonable choice when looking for a middle ground between recolors and number of colors used, on top of which they also scale well with both graph size and update sequence length, performing just as well when either of these increases.

Static-Dynamic Algorithm The static-dynamic algorithm is somewhat of an exception in that the idea behind it is intuitively quite smart, but the execution causes its performance to be less than desirable. The multiplicative growth in number of colors used caused by using two separate color palettes that need to later be combined makes this algorithm the least attractive choice in many situations. It is therefore not recommended to use this algorithm in any of the cases considered in this work.

DC-Orient While DC-Orient was always expected to be the go-to algorithm in cases where the total number of colors used are most important, the experiments have shown a whole new property of this algorithm. Where normally DC-Orient would require many

more recolors and time to run compared to other algorithms, this disadvantage seemingly disappears when ran on a highly skewed dataset. On datasets with unfair distribution of degrees or an ordered update sequence DC-Orient can compete with all other algorithms discussed in this work while still retaining the lowest number of colors used of them all.

Static-Simple The new static-simple algorithm has proven to be a strict improvement over the original static-dynamic algorithm. It manages to produce results with the same number of recolors, but a much lower number of total colors used. It also remains competitive with the small-bucket algorithm and the random warm-up, often outperforming the small-bucket algorithm and providing a reasonable trade-off in the range of low number of recolors. The static-simple algorithm does not manage to cover the same range as the big-bucket, DC-Orient or DC-Simple algorithms when it comes to low number of colors, however. Additionally, the random element of this algorithm, that becomes especially important for higher parameter values can make the results for this algorithm quite unpredictable, which is something to consider when deciding to use static-simple.

DC-Simple Possibly the most surprising and promising result of the experiments is the quality of the results produced by DC-Simple. This simple combination of the random warm-up and DC-Orient is seemingly able to simulate either one of them, but is also able to produce competitive results anywhere in the range between them. Similarly to DC-Orient, DC-Simple requires fewer recolors when the dataset is skewed somehow, but since the random-warm up algorithm displays no such behaviour, this effect is lessened with higher values for parameter p, in some cases, like the real-life Reddit dataset, it can thus occur that DC-Simple with some parameter p is outperformed by the original DC-Orient. The biggest drawback of DC-Simple remains its running time which, even though it considerably improves upon that of DC-Orient, is still not competitive with the alternative algorithms discussed in this work. This algorithm should thus only be used if the number of colors and recolors is sufficiently more important than the time it takes to run.

These conclusions indicate that, depending on the situation, almost all of the discussed algorithms are the best option in at least some cases. Aside from outlining what these cases are in order to improve decision making, we also introduced two new combination algorithms, proving that cooperation between multiple approaches can lead to results that surpass either approach by themselves. With this lesson and an extension to the available experimental data on the dynamic graph coloring problem, we hope this work will stimulate further research into this versatile problem.

Chapter 8

Future Work

Even though many experiments were performed over the course of the research for this work, the many variables make it impossible to run a test for each and every combination. Many of the algorithms have different versions available and most of them use a parameter to manage their trade-off. Apart from the options each algorithm provides, the graph and update sequence generation methods also allow for parameters to set the number of vertices, edges, the distribution of the degrees, maximum degree, order of updates and variation allowed in each of these. With all these options available there are many more experiments imaginable, in which a multitude of different combinations can be investigated.

Additionally, due to the limited access to time and computing power over the course of this work, an upper bound to the size of the experiments introduced itself. Future work might consider running experiments on even larger graphs, using the findings from this work as a guideline. As such, large graphs could be experimented on to find out if the algorithms deemed to scale better do indeed perform significantly better on an even larger scale. Apart from size, future work may also consider running experiments on graphs with even more significantly skewed aspects such as uneven degree distribution or node focused update sequence ordering. These inputs have proven to considerably change the relative performance of the different algorithms discussed in this work, and it would be interesting to see how strong this effect can become, as well as finding practical applications in which such skewed graphs naturally occur. Additionally, the parameter ranges used in Chapter 6 could be extended by considering more extreme values for the parameters. The plots could also be smoothed out even further by plotting more datapoints for each algorithm and taking the average of more instances for the randomized algorithms. More datapoints can also be computed for the Reddit dataset, allowing for further insight into the trade-off for these algorithms in a real-life application.

Another aspect that could prove interesting is the difference between measuring the average number of colors and recolors as compared to the maximum of these values. Since many of the algorithms perform an occasional reset step to reduce the number of colors used, the quality of the coloring for such an algorithm could vary quite significantly while still obtaining a reasonable average. Depending on the practical application this could be problematic behaviour. Future research could thus be conducted on which algorithms

most clearly display this behaviour, how it could affect a practical application, and whether or not a trade-off may be possible between stability and quality.

Apart from these additional experiments, the combination algorithms presented in this thesis could also be extended upon. While both of the new algorithms introduced in this work are seemingly competitive with many of the original algorithms, no theoretical analysis has been performed to prove asymptotic bounds for either of the two algorithms. The conclusions in this work are merely based on an intuitive understanding of each algorithm and the experiments that were performed. As such, different methods of combining the original algorithms may exist that perform even better than those found here, or improvements upon these algorithms might exist that allow them to perform even more efficiently with only minor changes. These new algorithms, but especially the idea of combining existing algorithms, is therefore an interesting topic for further research.

Bibliography

[1] M. Theunis. (2022) Dynamic graph coloring thesisx. *GitHub repository*. Available: https://github.com/mtheunistue/DynamicGraphColoring

[2] P. M. Pardalos, T. Mavridou, and J. Xue, *The Graph Coloring Problem: A Bibliographic Survey*. Boston, MA: Springer US, 1998, pp. 1077–1141. ISBN 978-1-4613-0303-9. doi: 10.1007/978-1-4613-0303-9_16

[3] R. M. Karp, *Reducibility among Combinatorial Problems*. Boston, MA: Springer US, 1972, pp. 85–103. ISBN 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9

[4] M. Garey, D. Johnson, and L. Stockmeyer, "Some simplified np-complete graph problems," *Theoretical Computer Science*, vol. 1, no. 3, 1976, pp. 237–267. doi: 10.1016/0304-3975(76)90059-1

[5] L. Stockmeyer, "Planar 3-colorability is polynomial complete," *SIGACT News*, vol. 5, no. 3, jul 1973, p. 19–25. doi: 10.1145/1008293.1008294

[6] R. Lewis, J. Thompson, C. Mumford, and J. Gillard, "A wide-ranging computational comparison of high-performance graph colouring algorithms," *Computers Operations Research*, vol. 39, no. 9, 2012, pp. 1933–1950. doi: 10.1016/j.cor.2011.08.010

[7] A. Kosowski and K. Manuszewski, "Classical coloring of graphs," *Graph Colorings*, 2004, p. 1–19. doi: 10.1090/conm/352/06369

[8] D. J. A. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, 01 1967, pp. 85–86. doi: 10.1093/comjnl/10.1.85

[9] S. Vishwanathan, "Randomized online graph coloring," *Journal of Algorithms*, vol. 13, no. 4, 1992, pp. 657–669. doi: 10.1016/0196-6774(92)90061-G

[10] L. Ouerfelli and H. Bouziri, "Greedy algorithms for dynamic graph coloring," in *2011 International Conference on Communications, Computing and Control Applications (CCCA)*, 2011, pp. 1–5. doi: 10.1109/CCCA.2011.6031437

[11] J. Bossek, F. Neumann, P. Peng, and D. Sudholt, "Runtime analysis of randomized search heuristics for dynamic graph coloring," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1443–1451. ISBN 9781450361118. doi: 10.1145/3321707.3321792

[12] S. Bhattacharya, D. Chakrabarty, M. Henzinger, and D. Nanongkai, *Dynamic Algorithms for Graph Coloring*, pp. 1–20. doi: 10.1137/1.9781611975031.1

[13] L. Barba, J. Cardinal, M. Korman, S. Langerman, A. van Renssen, M. Roeloffzen, and S. Verdonschot, "Dynamic graph coloring," *Algorithmica*, vol. 81, no. 4, Apr. 2019, pp. 1319–1341. doi: 10.1007/s00453-018-0473-y

[14] S. Solomon and N. Wein, "Improved dynamic graph coloring," *ACM Trans. Algorithms*, vol. 16, no. 3, jun 2020. doi: 10.1145/3392724

[15] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "Effective and efficient dynamic graph coloring," *Proceedings of the VLDB Endowment*, vol. 11, 11 2017, pp. 338–351. doi: 10.14778/3157794.3157802

[16] A. Hagberg, D. Schult, and P. Swart. Networkx reference release 2.8.4. *NetworkX*. Available: https://networkx.org/documentation/stable/_downloads/networkx_reference.pdf

[17] S. Kumar, W. L. Hamilton, J. Leskovec, and D. Jurafsky, "Community interaction and conflict on the web," in *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2018, pp. 933–943. doi: 10.1145/3178876.3186141

Appendix A

Comparison Tables

This appendix consists of experimental results obtained during preliminary experimentation on the various algorithms to find out which versions to focus on. The results in this appendix are provided in the form of tables in which the version of the algorithm, average number of recolors per update, average number of colors used in the coloring after each update and total time taken for all updates are provided as columns. Additionally, in the top left cell of each table, some more information is provided about the graph used in the preliminary experiment. The number of nodes, edges and average estimate for the chromatic number C after each update are given. The experiments provided here follow the same set-up as described in Section 6, we thus consider increasing graphs and stream experiments.

A.1 Random Warm-Up Version Comparison

In Tables A.1 and A.2 we provide a small comparison of the different implementations of the random warm-up algorithm. The version named 'Random Warm-Up (local degrees)' is the one used in the main experiments.

Table A.1: Comparison of the random warm-up algorithms on a random graph and update sequence.

250 Nodes; 23623 Edges; C = 32.14	Average nr. of Recolors	Average nr. of Colors	Time Taken (s)
Random Warm-Up (local degrees)	0.04	82.11	3.687
Random Warm-Up 2 (local degrees)	0.07	84.05	3.851
Random Warm-Up (max degree)	0.01	164.51	3.854
Random Warm-Up 2 (max degree)	0.02	164.17	3.909

Table A.2: Comparison of the random warm-up algorithms on a random update stream of length 10.000.

161 Nodes; 4043 Edges; C = 18.72	Average nr. of Recolors	Average nr. of Colors	Time Taken (s)
Random Warm-Up (local degrees)	0.01	53.7	1.012
Random Warm-Up 2 (local degrees)	0.03	52.95	0.991
Random Warm-Up (max degree)	0.01	80.79	0.951
Random Warm-Up 2 (max degree)	0.01	82.55	0.947

As can be seen in both tables, the difference in performance between the random warm-up result 2 as described in the paper by Bhattacharya et al. [12] and the combined random warm-up algorithm are fairly insignificant and mostly due to the randomness involved. When comparing the results between using local degrees or a global maximum degree however, it is clear that the local degree version has a bias towards using fewer colors in total but recoloring more often as compared to the global maximum degree. This result is intuitively correct, since making each color palette the size of the maximum degree allows for more colors to be chosen, increasing the total of colors but reducing the number of conflicts. It is thus worth noting that an additional trade-off for the random warm-up algorithm is possible by using the maximum degree based version instead of the one used in the main experiments.

A.2 Static-Dynamic Version Comparison

In Tables A.3 and A.4 we provide a small comparison between the two different implementations for the static-dynamic algorithm: one with and one without resetting the dynamic graph whenever a full reset is called. The version without a dynamic reset is the one presented in [14] and the main version considered in this work.

Table A.3: Comparison of the static-dynamic algorithm with and without dynamic reset on random graph and update sequence.

235 Nodes; 18528 Edges; C = 27.17	Average nr. of Recolors	Average nr. of Colors	Time Taken (s)
Static-Dynamic (l = 10) no dynamic reset	1.06	132.49	6.902
Static-Dynamic (l = 10) dynamic reset	1.19	108.14	6.929
Static-Dynamic (l = 100) no dynamic reset	0.18	163.82	6.158
Static-Dynamic (l = 100) dynamic reset	0.2	155.13	6.494

Table A.4: Comparison of the static-dynamic algorithm with and without dynamic reset on a random update stream of length 10.000.

207 Nodes; 7315 Edges; C = 24.54	Average nr. of Recolors	Average nr. of Colors	Time Taken (s)
Static-Dynamic (l = 10) no dynamic reset	0.51	124.94	2.513
Static-Dynamic (l = 10) dynamic reset	0.56	108.34	2.814
Static-Dynamic (l = 100) no dynamic reset	0.11	157.7	2.421
Static-Dynamic (l = 100) dynamic reset	0.11	157.13	2.336

From these results it appears the dynamic reset variant provides a slight skew towards fewer total colors used and more recolors. While this could be an interesting trade-off, especially since the effect on number of colors used seems to be much larger, it is not significant enough to make this algorithm competitive with the others considered in this thesis. We therefore decide to use the original version as described in [14] in the rest of the work, in order to get a clearer comparison between the algorithms from the different papers.

A.3 DC-Orient Version Comparison

A comparison between the performance of the basic and optimized versions of DC-Orient can be found in Tables A.5 and A.6.

Table A.5: Comparison of the basic and optimized versions of DC-Orient on a random graph and update sequence.

178 Nodes; 6332 Edges; C = 14.3	Average nr. of Recolors	Average nr. of Colors	Time Taken (s)
DC-Orient without optimizations	17.22	14.3	108.322
DC-Orient with optimizations	17.22	14.3	21.71

Table A.6: Comparison of the basic and optimized versions of DC-Orient on a random update stream of length 10.000.

225 Nodes; 13645 Edges; C = 40.06	Average nr. of Recolors	Average nr. of Colors	Time Taken (s)
DC-Orient without optimizations	38.3	40.06	2073.542
DC-Orient with optimizations	38.3	40.06	172.858

As is apparent from these results, both versions of DC-Orient are identically effective when it comes to recolors and total number of colors used. The only difference between the two lies in the running time. Because of this, the optimized DC-Orient version is used in the rest of this work. It is also worth noting that indeed the number of colors used by DC-Orient are identical to those used by the static greedy approach used to approximate C, as is apparent from these values being identical in both experiments.

A.4 DC-Simple Version Comparison

Tables A.7 and A.8 show the difference in performance between the basic and optimized versions of DC-Simple.

Table A.7: Comparison of the basic and optimized versions of DC-Simple on a random graph and update sequence.

231 Nodes; 8383 Edges; C = 14.25	Average nr. of Recolors	Average nr. of Colors	Time Taken (s)
Basic DC-Simple (p = 0.8)	5.63	16.27	52.536
Optimized DC-Simple (p = 0.8)	5.47	17.04	13.881
Basic DC-Simple (p = 0.998)	0.2	30.93	6.145
Optimized DC-Simple (p = 0.998)	0.25	30.03	5.307

Table A.8: Comparison of the basic and optimized versions of DC-Simple on a random update stream of length 10.000.

225 Nodes; 9148 Edges; C = 27.37	Average nr. of Recolors	Average nr. of Colors	Time Taken (s)
Basic DC-Simple (p = 0.8)	9.32	27.76	277.483
Optimized DC-Simple (p = 0.8)	9.24	27.86	39.038
Basic DC-Simple (p = 0.998)	0.26	38.2	17.299
Optimized DC-Simple (p = 0.998)	0.22	42.37	11.105

A similar result as for DC-Orient can be observed here: the results of both versions are very similar while the running time of the optimized version is much lower, especially for the iteration with p = 0.8. The reason the results are not identical for the different versions is that the random warm-up element included in DC-Simple causes an element of randomness to affect the final results, it is therefore highly unlikely that even the same algorithm produces the exact same result twice. We thus also use the optimized version of DC-Simple in the main experiments.

A.5 Parameter Comparison

Table A.9: The results of the algorithms ran using various parameters on the same graph and update sequence as Figure 6.3.

200 Nodes; 11940 Edges; C = 21.5	Average nr. of Recolors	Average nr. of Colors	Time Taken (s)
Random Warm-Up	0.05	54.64	1.521
Small-Bucket algorithm (d = 1)	0.31	175.12	4.427
Small-Bucket algorithm (d = 3)	3.26	44.35	3.326
Small-Bucket algorithm (d = 5)	3.92	40.9	3.75
Small-Bucket algorithm (d = 10)	4.39	45.37	4.793
Small-Bucket algorithm (d = 20)	4.39	45.43	5.356
Big-Bucket algorithm (d = 1)	24.64	21.5	14.318
Big-Bucket algorithm (d = 3)	6.97	26.19	4.267
Big-Bucket algorithm (d = 5)	5.25	30.16	3.773
Big-Bucket algorithm (d = 10)	4.47	39.34	5.205
Big-Bucket algorithm (d = 20)	4.47	39.34	5.206
Static-Dynamic algorithm (l = 1)	4.9	64.88	5.885
Static-Dynamic algorithm (l = 2)	3.33	83.26	4.906
Static-Dynamic algorithm (l = 5)	1.77	100.34	4.086
Static-Dynamic algorithm (l = 10)	1.04	108.64	3.803
Static-Dynamic algorithm (l = 50)	0.29	130.59	3.274
Static-Dynamic algorithm (l = 100)	0.17	127.95	3.141
DC-Orient	24.64	21.5	83.747
Static-Simple algorithm (l = 1)	4.76	34.88	5.76
Static-Simple algorithm (l = 2)	3.09	38.33	4.349
Static-Simple algorithm (l = 5)	1.56	42.32	3.052
Static-Simple algorithm (l = 10)	0.89	45.55	2.453
Static-Simple algorithm (l = 50)	0.24	56.7	1.674
Static-Simple algorithm (l = 100)	0.14	60.37	1.629
DC-Simple (p = 0.2)	20.22	21.96	71.565
DC-Simple (p = 0.5)	14.23	22.38	54.849
DC-Simple (p = 0.8)	6.84	23.63	31.353
DC-Simple (p = 0.998)	0.31	40.33	10.059
DC-Simple (p = 0.9999)	0.06	54.5	8.768

Appendix B

Pseudocode

In this appendix, the pseudocode for the small-bucket algorithm, the big-bucket algorithm and the static-dynamic algorithm are provided. This pseudocode represents the manner in which the algorithms have been implemented during this thesis and may thus vary slightly from the algorithms as described in the original papers [13] [14]. The textual description of the small- and big-bucket algorithms can be found in Section 4.3 and the description of the static-dynamic algorithm in Section 4.4.

Algorithm 1 Small-Bucket Algorithm

Parameters: d, G
Initialization:
resetBuckets(G)
Return;

Update graph:
if Edge or Vertex removed **then**:
    Update all relevant subgraphs without changing colors
if Vertex added **then**:
    Add vertex to an empty bucket b on the first level
    updateBuckets(b);
if Edge added **then**:
    Choose one of the endpoints of the edge to be v_e
    Remove v_e from whichever bucket it is in and add it to an empty bucket b on the first level
    updateBuckets(b);

updateBuckets(b):
$i := 0$;
while $i < d$ **do**
    if Still an empty bucket at level i **then**:
        Let b_g denote the subgraph of the nodes in b
        staticColoring(b_g);
        Return;
    else:
        Empty all level i buckets into a single level i+1 bucket, update b to point at the new bucket
        i++;
resetBuckets(G);
Return;

resetBuckets(G):
$N_R :=$ number of vertices in G
$s := N_R^{1/d}$
Create d levels of s buckets each, having s^i capacity for level i, starting at 0
Create a final level d with a single bucket without a limit on capacity
staticColoring(G);
Return;

staticColoring(g):
Use the available static graph coloring algorithm to color subgraph g
Return;

Algorithm 2 Big-Bucket Algorithm

Parameters: d, G
Initialization:
resetBuckets(G)
Return;

Update graph:
if Edge or Vertex removed **then**:
    Update all relevant subgraph without changing colors
if Vertex added **then**:
    Add vertex to bucket b on the first level
    updateBuckets(b);
if Edge added **then**:
    Choose one of the endpoints of the edge to be v_e
    Remove v_e from whichever bucket it is in and add it to bucket b on the first level
    updateBuckets(b);

updateBuckets(b): $i := 1$;
while $i < d + 1$ **do**
    if Less than or equal to $s^i - s^{i-1}$ vertices in b **then**:
        Let b_g denote the subgraph of the nodes in b
        staticColoring(b_g);
        Return;
    else:
        Empty b into the level i+1 bucket, update b to point at the new bucket
        i++;
resetBuckets(G);
Return;

resetBuckets(G):
$N_R :=$ number of vertices in G
$s := N_R^{1/d}$
Create d levels of a single bucket each, having s^i capacity for level i, starting at 1
Create a final level $d + 1$ with a single bucket without a limit on capacity
staticColoring(G);
Return;

staticColoring(g):
Use the available static graph coloring algorithm to color subgraph g
Return;

Algorithm 3 Static-Dynamic Algorithm for General Graphs

Parameters: l, G
Initialize:
let n be the number of vertices in G
$c := 0$; Counter variable
staticBlackBox(G, 0);
let G' be G without edges
Initialize all colors in G' as 0
let $r_0 ... r_{\log n}$ be an empty set of nodes
let R(r) be a subgraph of G depending on a selection of nodes r
Activate levels 0..$\log n$;

Update graph:
if Edge or Vertex removed **then**:
    Update G and G' without changing colors
if Vertex added **then**:
    Add vertex to G and G'
    Assign new vertex arbitrary colors c_1 and c_2
if Edge added **then**:
    Let e be the added edge
    Increase recent degree of endpoints by 1
    Add edge to G and G'
    Add node with the highest recent degree to r_{level} for each active level
$c := c + 1 \bmod l$;
if $c \bmod l == 0$ **then**:
    Set *level* to be the highest of the active levels
    if *level* $== 0$ **then**:
        staticBlackBox(G, *level*)
    else
        staticBlackBox(R(r_{level}), *level*);
        Deactivate *level*
    Activate all levels higher than *level*
    Return;
if Edge added **then**:
    if e still in G' **then**:
        dynamicBlackBox(G', e);
Return;

staticBlackBox(g, *level*):
Compute valid coloring for g using the colors from *level*
Assign vertices in g the computed color c_1
Reset recent degree for all vertices in g to 0
Remove all edges adjacent to vertices in g from G'
Remove all nodes from r_{level}
Return;

dynamicBlackBox(g, e):
Compute valid coloring for nodes adjacent to e in g
Assign vertices in g the computed color c_2
Return;